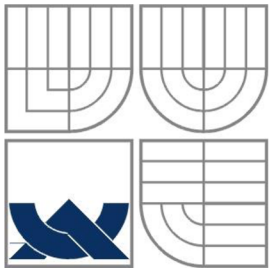
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

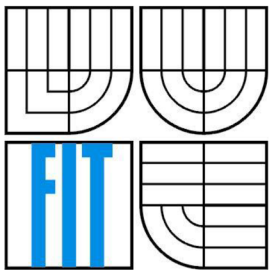
Fakulta informačních technologií
Faculty of Information Technology

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

Brno, 2016 Marek Majcher



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

3D HRA V UNITY S VYUŽITÍM ANALÝZY HUDBY

3D GAME IN UNITY WITH USAGE OF MUSIC ANALYSIS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MAREK MAJCHER

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ MILET

BRNO 2016

Abstrakt

Bakalářská práce se zabývá metodami analýzy zvukových signálů, enginem Unity a procedurálním generováním grafiky. Cílem práce bylo vytvoření funkční hry, která by demonstrovala analýzu hudby na vhodném způsobu procedurálního generování grafiky. Důležitým grafickým médiem, které se spolupodílí na výsledném grafickém efektu, jsou různé grafické shadery, které dotváří výsledný audiovisuální efekt a dělají vzhled příjemný pro hráče.

Abstract

This bachelor thesis deals with various methods of analysis of sound signals, Unity engine and procedural generation of graphics. The main goal was to create a working game that would demonstrate music analysis in an appropriate way using procedural generation of graphics. Important graphical medium, that collaborates on final graphical effect, are various graphic shaders that mold final audiovisual effect thus making it interesting for the player.

Klíčová slova

Unity engine, analýza zvuku, shader, vizuální efekty, procedurální generovíní

Keywords

Unity engine, sound analysis, shader, visual effects, procedural generation

Citace

Majcher Marek: 3D Hra v Unity s využitím analýzy hudby, bakalářská práce, Brno, FIT VUT v Brně, 2016

3D Hra v Unity s využitím analýzy hudby

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……

Marek Majcher
18. 5. 2016

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Tomášovi Miletovi za podporu a rady při vytváření této práce.

Obsah

1 Úvod

V priebehu posledných pár rokov došlo k výraznému priblíženiu tvorby hier k užívateľom, keďže nástroje na ich tvorbu sa stali intuitívnejšie, robustnejšie a jednoduchšie na používanie. Okrem enginov, používaných veľkými spoločnosťami, sa práve v poslednom období dostal na trh aj engine Unity, ktorý je obľúbený najmä v sfére nezávislej tvorby.

Možnosti týchto enginov sú rozsiahle a umožňujú autorom testovať naplno hranice ich fantázií, pričom umožňujú implementáciu najrôznejších štýlov a herných mechanizmov.

V dnešnej dobe sa na trhu s veľkými AAA hernými projektmi vyskytujú najčastejšie hry, implementujúce pomerne jednoduché, časom overené herné mechanizmy ako napríklad RPG hry, strategické hry, FPS alebo adventúry.

Naopak v oblasti nezávislej tvorby sú programátori a dizajnéri nútení prichádzať na trh s novými a inovatívnymi nápadmi, ktoré by zaujali dostatočne veľkú cieľovú skupinu na trhu a zabezpečili si tak ekonomický profit. Výsledkom sú projekty, ktoré rôznymi spôsobmi implementujú ovládanie gestami, respektíve sú založené na zvuku (melódii, respektíve iných aspektoch zvuku) alebo sa zas sústreďujú na netradičné implementovanie príbehu do rôznych logických hádaniek, kde príbeh nemusí byť nutne vyrozprávaný, ale je možné ho vyčítať zo samotných herných mechanizmov.

Cieľom tejto práce je vytvorenie hry v hernom engine Unity, založenej na spracovaní zvuku a procedurálnom generovaní obsahu na základe rôznych aspektov analýzy vloženého hudobného súboru. Základom hry bude mechanizmus vesmírnej prestrelky, v ktorej bude úlohou hráča vyhýbať sa a zároveň ničiť proti nemu idúce asteroidy, ktoré budú generované na základe už spomenutej analýzy, pričom ich povrch (a jeho deformácia) bude ovplyvňovaný nielen aktuálne prehrávanou hudbou, ale aj funkciou šumu (v tomto prípade jej simplex variantov).

2 Teória

Táto kapitola pokrýva základné teoretické vedomosti využité pri tvorbe tejto práce, pričom teória špecifická pre herný engine Unity je popísaná v kapitole 3. V tejto kapitole hovorím o oblastiach, ktorých sa táto práca týka, a vysvetľujem základné pojmy (a ďalšie potrebné teoretické vedomosti a znalosti).

2.1 Herný Engine (herné jadro)

Herný engine (priblížený v dokumente [1]) je framework navrhnutý na tvorbu videohier. Základná funkcionalita, ktorú engine poskytuje, zahŕňa renderer pre 2D alebo 3D grafiku, fyzikálny engine alebo detekciu kolízií, spracovanie zvuku, skriptovanie, animácie, umelú inteligenciu a sieťovú komunikáciu. Všetky tieto prvky dohromady vytvárajú integrované vývojárske prostredie, ktorého účelom je umožniť vývojárom zjednodušený a hlavne rýchlejší vývoj hier v data-driven zmysle.

Snahou vývojárov enginu je vytvorenie robustného softvéru, ktorý obsahuje viacero nástrojov, ktoré môžu herní vývojári potrebovať na vytvorenie hry. Herný engine, tak ako je tu popísaný, sa často označuje aj ako middleware, pretože ponúka flexibilnú a opakovane použiteľnú softvérovú platformu, ktorá zároveň poskytuje plnú základnú funkcionalitu, potrebnú na vývoj herných aplikácií a ktorá zároveň redukuje náklady, komplikácie a čas do uvedenia vyprodukovaných herných aplikácií na trh – toto všetko je považované za kritické faktory, potrebné na úspech v dnešnom konkurenčnom hernom priemysle.

Podobne ako iné middleware riešenia, herné enginy ponúkajú platformovú abstrakciu, čím umožňujú, aby tá istá hra mohla byť plne funkčná na viacerých platformách (konzoly, PC, atď.), prípadne len s minimálnymi úpravami zdrojového kódu. Často sú herné enginy navrhované s architektúrou, zameranou na komponenty, ktorá umožňuje nahradzovanie špecifických súčastí systému enginu špecializovanými komponentami ako je napríklad Havok pre fyziku alebo Bink pre video. Aj napriek svojmu menu sa herné enginy nevyužívajú len na tvorbu hier, ale aj na tvorbu rôznych softvérových demo verzií v rôznych oblastiach použitia, napr. v architektúre, pri tréningových simuláciách alebo pri modelovaní rôznych prostredí.

Najčastejšie sú 3D enginy postavené na grafickom API, ako je napríklad Direct3D alebo OpenGL, ktoré poskytujú softvérovú abstrakciu GPU. Často sa používajú aj low-level knižnice ako DirectX, Simple DirectMedia Layer a OpenAl, keďže ponúkajú na hardvéri nezávislý prístup k ostatnému hardvéru napríklad k vstupným zariadeniam, sieťovým kartám a zvukovým kartám.

2.2 Preprocessing versus Real-time spracovanie

Na začiatku riešenia tohto projektu bolo potrebné prijať závažné rozhodnutie o výbere jedného z dvoch možných variantov riešenia projektu, ktorými sú: preprocessing alebo spracovanie v reálnom čase. Každý variant má svoje pozitíva aj negatíva a každý variant určí iné (celkové) smerovanie projektu.

Preprocessing znamená, že pred spustením scény dôjde k spracovaniu všetkých dát požadovaných danou scénou a až po spracovaní všetkých dát dôjde k spusteniu scény. Tento prístup by znamenal, že na začiatku preskúmam zvukový súbor, poznamenám si všetky hľadané aspekty (rytmus, výšky atď.), ich hodnoty a časový výskyt. Na základe týchto dát je potom možné vygenerovať samotný level, ktorý presne zodpovedá prehrávanej piesni. Nevýhodou tohto prístupu je nutnosť zabezpečiť synchronizáciu medzi aktuálne prehrávanou hudbou a aktuálnym stavom scény.

Očividnou nevýhodou je nutnosť čakať na spracovanie levelu pred jeho spustením, keďže tento čas narastá s dĺžkou a komplexnosťou piesne.

Real-time spracovanie (alebo spracovanie v reálnom /skutočnom čase) znamená, že scéna sa načíta tak, ako je, pričom požadované dáta sú spracovávané počas priebehu tejto scény. Tento prístup znamená, že scéna je pripravená takmer okamžite, ako vznikne požiadavka na jej spustenie. Nevýhodou tohto prístupu je fakt, že má vyššie požiadavky na procesný výkon, keďže dáta sú spracovávané v priebehu scény. Procesor však v tomto prípade nemusí spracovávať všetky potrebné dáta, keďže v jednom okamihu je potrebné spracovať len aktuálnu zvukovú stopu. Ak projekt vyžaduje, aby behom každého framu došlo k spracovaniu veľkého množstva dát alebo vykonaniu vysoko komplexných výpočtov nad týmto balíkom dát, tak je možné, že vo výsledku bude hodnota FPS (frames per second) príliš nízka na to, aby bolo možné scénu ovládať. V takomto prípade existujú dve možné riešenia, a to buď redukcia množstva dát, ktoré analyzujeme a čím zjednodušíme výpočty, potrebné na spracovanie dát, alebo rozdelenie spracovávaných dát a výpočtov na menšie úlohy, ktoré pridelíme viacerým procesorom.

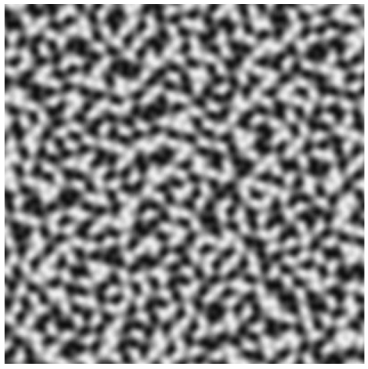
Bežne sa v reálnych projektoch nepoužíva iba čistý preprocessing alebo spracovanie úplne v reálnom čase, ale kombinácia oboch prístupov. Časť dát je vytvorená už na začiatku a pripravená na zobrazenie, pričom zvyšok scény je dotvorený až v momente vzniku požiadavky. Takto je možné vyvažovať požiadavky na výkon a čas potrebný na pripravenie scény.

V tomto projekte som sa zameral na druhý prístup – spracovanie v reálnom čase. Keďže paralelné spracovanie (threading) nebolo jadrom mojej práce, tak som sa rozhodol znížiť množstvo výpočtov, a tým uvoľniť výkon pre ostatné výpočty.

2.2.1 Procedurálne generovanie

Najbežnejším príkladom spracovania v reálnom čase je procedurálne generovanie. Tento prístup sa zakladá na vytváraní obsahu prostredníctvom algoritmov. Priebeh týchto algoritmov môže byť kontrolovaný prostredníctvom rôznych vstupov, (ktoré môžu byť ešte upravené pomocou funkcií sínu, kosínu alebo iných). Ako vstup je možné použiť šumové funkcie (perlin/simplex noise), video alebo audio vstupy.

V tomto projekte sa používa 4D simplex noise textúra (obrázok 2.1), ktorá sa mení s časom a ovplyvňuje grafický obsah levelu, pričom zvukový vstup je využívaný na kontrolovanie tempa hry. V tomto projekte som využil simplex noise funkciu popísanú v dokumente [2].



Obrázok 2.1: Príklad noise textúry

2.3 Spracovanie zvuku

Táto podkapitola pokrýva teóriu spojenú so spracovaním aktuálne prehrávaného zvuku do takej podoby, z ktorej sme schopný vyextrahovať parametre použiteľné v samotnej hre.

2.3.1 Rýchla Fourierova transformácia (FFT)

Rýchla Fourierova transformácia (informácie prebrané z dokumentu [3]) sa používa na vypočítanie diskrétnej Fourierovej transformácie (a inverznej DFT). Fourierova analýza konvertuje čas na frekvenciu (a naopak); rýchla Fourierova transformácia vypočíta túto transformáciu faktorizáciou DFT matice na súčet riedkych (najčastejšie nulových) faktorov.

FFT rieši DFT (vzorec 2.1) a produkuje tie isté výsledky ako priame vyhodnocovanie DFT definície, pričom hlavným a najdôležitejším rozdielom je, že FFT je oveľa rýchlejšia (mnohé FFT algoritmy sú dokonca presnejšie ako priame vyhodnocovanie DFT definície). Vzorec 2.1 pre DFT má kvadratickú náročnosť, keďže máme N výstupov a každý výstup je súčtom N termov, pričom FFT okienko redukuje túto náročnosť na linearitmickú. Samotné unity sa postará nielen o výpočet FFT, ale zároveň dodáva aj orezávacie okienkové funkcie: obdĺžnikovú, trojuholníkovú, Hammingovu, Hanningovu, Blackmanovu a BlackmanHarrisovu. Tieto funkcie sa používajú na časové alebo frekvenčné orezanie (pokiaľ nás zaujíma iba krátke časové obdobie alebo iba určitý úsek frekvencií).

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \qquad k = 0, \dots, N-1 \tag{2.1}$$

2.3.2 Vlnková transformácia

Vlnková transformácia (bližšie popísaná v knihe [4], kapitole 4) je jednou z najpopulárnejších časovo-frekvenčných transformácií, pričom umožňuje rozložiť signál na nezávislé stavebné kamene. Základnou myšlienkou vlnkovej transformácie je, že by mala umožňovať len zmenu času (časového rozsahu), nie tvaru funkcie. Toto je dosiahnuté výberom správnej základnej funkcie, ktorá toto umožňuje.

Prakticky sa zoberie skonštruovaná vlnka (napríklad vlnka reprezentujúca tón C), a tá sa posúva po testovanom signáli, pričom nám korelácia medzi vlnkou a testovaným signálom hovorí o tom, nakoľko sa daná vzorka podobá na hľadanú vzorku (napríklad na ten tón C).

$$y_{low}[n] = \sum_{k=-\infty}^{\infty} f[k] h[2n-k] \tag{2.2}$$

$$y_{high}[n] = \sum_{k=-\infty}^{\infty} f[k] g[2n-k] \tag{2.3}$$

Pri diskrétnej vlnkovej transformácii prechádza signál dvoma (pri kaskádovaní aj viacerými) sadami filtrov. Vzorce 2.2 a 2.3 sú vzorcami pre výpočet približných koeficientov (vzorec 2.2) a podrobných koeficientov (obr. 2.3). Tak, ako sú tieto filtre navrhnuté, umožňujú perfektnú rekonštrukciu signálu (sú k sebe komplementárne) a označujú sa ako kvadratúrne zrkadlové filtre.

2.4 Detekcia rytmu

Základným kameňom danej práce je práve detekcia rytmu (obsah kapitoly založený na informáciách z dokumentu [5]). Základným problémom danej oblasti je fakt, že pre ľudí a zvieratá je vnímanie rytmu prirodzené - ľudia prirodzene cítia rytmus, na ktorý následne reagujú. Otázkou ostáva, ako stroj naučiť rozoznať to, čo my ľudia cítime. Z dôvodu potreby zabezpečenia čo najrealistickejšieho efektu je teda potrebné do stroja vložiť ľudské vnímanie - človek napríklad nepotrebuje si vypočuť celú pieseň, aby určil úder, a teda ani stroj nepotrebuje k analýze celú zvukovú stopu, ale postačí iba lokálna oblasť miesta, na ktorom testujeme prítomnosť úderu.

2.4.1 Detekcia pomocou energie zvuku

Ako už bolo spomenuté v kapitole 2.4, tak pre detekciu rytmu nie je potrebná celá zvuková stopa, ale stačí aj lokálna oblasť - detekujeme úder iba vtedy, keď je aktuálna energia zvuku vyššia ako priemerná lokálna energia. Až pri samotnom výpočte sa však ukáže hlavný rozdiel medzi detekciou jednoduchých signálov a rytmu v hudbe. Niektoré konštanty je totiž možné len odhadnúť - napríklad dĺžku stopy, ktorá bude reprezentovať lokálne miesto, z ktorého budeme prepočítavať hodnotu priemernej lokálnej energie. V tomto prípade sa najčastejšie používa hodnota 1 sekunda - to je kompromis medzi spočítavaním príliš veľkého množstva dát, ktoré môže mať negatívny vplyv na výslednú analýzu, a príliš malým množstvom dát, ktoré môže vyprodukovať príliš nepresný výsledok.

Tento prístup je veľmi presný a znie správne pre hudobné štýly techno a rap, ale je to spôsobené aj tým, že dané štýly obsahujú jasný rytmus a väčšinou len malé množstvá šumu. V prípade použitia tohto spôsobu detekcie na iné štýly je veľká šanca, že dosiahnuté výsledky budú viac než nepresné.

2.4.2 Detekcia na základe frekvencie

Hlavným problémom s predchádzajúcim spôsobom detekcie rytmu je fakt, že detekcia pomocou energie zvuku je farboslepá voči jednotlivým frekvenčným subpásmam. V bežných pesničkách je normálne, keď rozoznávame rytmus pre rôzne hudobné nástroje ako napríklad bicie alebo gitara. Detekcia na základe energie zvuku však neberie ohľad na frekvenčné subpásmo a teda, ak dochádza k striedaniu úderov medzi bicími nástrojmi a gitarou, tak analyzér z toho detekuje relatívne pravidelnú vysokú hladinu energie zvuku, a teda nedôjde k (takmer) žiadnej detekcii. Ak chceme detekovať rytmus v komplikovaných a komplexných piesňach, ktoré využívajú rôzne nástroje, tak je potrebné zvoliť prístup, ktorý berie do úvahy energiu na danej frekvencii - prakticky energii na danom frekvenčnom subpásme.

Základným princípom tejto metódy je preberanie vzoriek zvuku, pričom až ich nazbierame 1024, tak pomocou rýchlej Fourierovej transformácie získame spektrum, ktoré potom rozdelíme na toľko subpásiem, koľko len potrebujeme. Tu je hlavným problémom konflikt medzi vysokým počtom subpásiem - čo produkuje presný výsledok, ale má vysokú výpočtovú záťaž - a nízkym počtom subpásiem - čo má nízku výpočtovú záťaž, ale produkuje nepresný výsledok. Zvyšok postupu je podobný ako v predchádzajúcom postupe, no rozdiel je vo výsledku. Teraz máme toľko lokálnych priemerov, koľko máme subpásem a v každom z týchto subpásem môžeme detekovať rytmus.

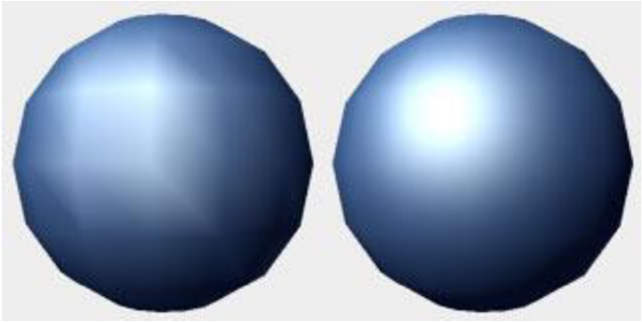
2.5 Použité (Unity) shadre

Existuje veľké množstvo rôznych druhov shadrov, kde každý plní inú špecifickú úlohu. Už len jedna kategória poskytuje veľké možnosti toho, čo je možné s daným shadrom dosiahnuť. V tomto projekte som sa rozhodol demonštrovať použitie niekoľkých, často používaných shadrov, ako sú vertex/fragment shader, geometry shader a tessalation shader.

2.5.1 Vertex/Fragment Shader

Vertex shadre sú najbežnejším druhom 3D shadrov. Vertex shader zbehne raz pre každý vertex, ktorý je poslaný do grafického procesora. Tieto shadre pracujú s pozíciou, farbou a súradnicou textúry. Poskytujú veľkú kontrolu nad pozíciou, pohybom, osvetlením a farbou.

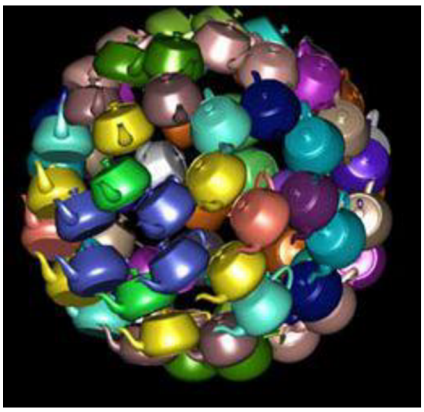
Fragment shader vypočítava farbu a ostatné atribúty pre každý fragment (pixel). Samotný fragment shader neprodukuje veľmi komplexné výsledky, keďže pracuje len s jedným fragmentom a nemá informácie o geometrii scény. Tento shader má/môže mať informáciu o pozícii na obrazovke, kde sú jednotlivé body zobrazované, čo umožňuje širokú škálu postprocessing efektov. Na obrázku 2.1 je vidieť porovnanie týchto dvoch prístupov.



Obrázok 2.1: Gouraud vs phong shading

2.5.2 Geometry Shader

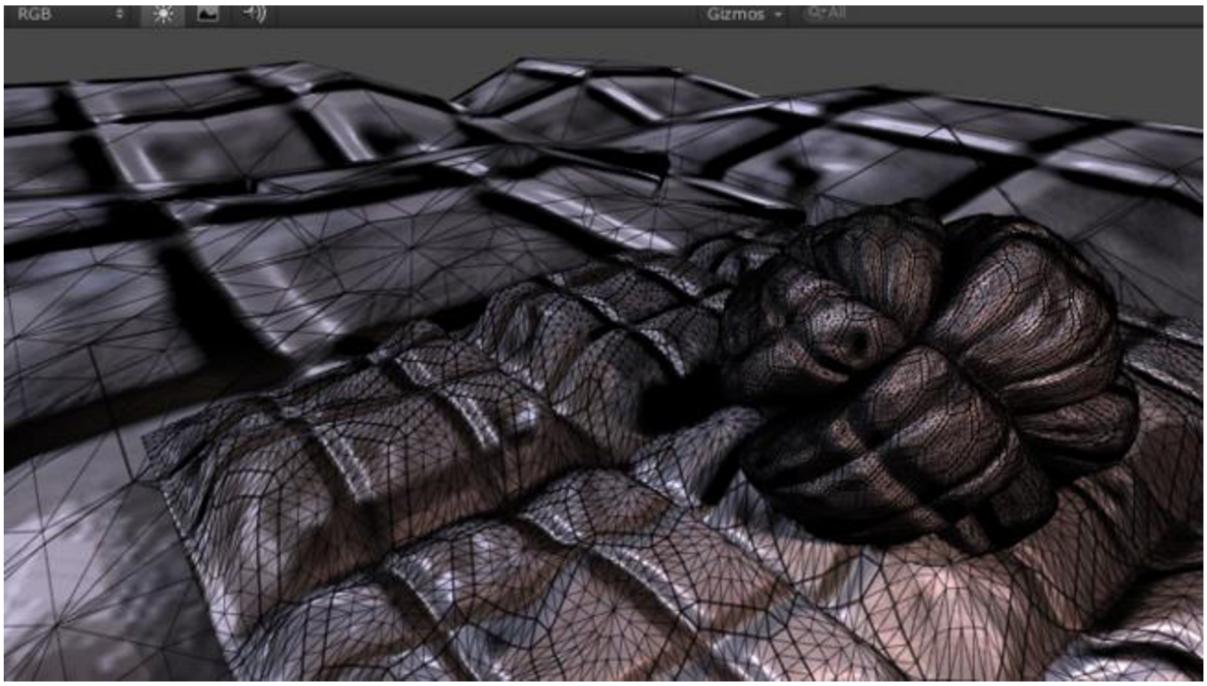
Geometry shadre sú zvláštnym typom shadra, ktorý dokáže generovať nové grafické primitíva, ako sú body, hrany a trojuholníky. Tento shader je spracovaný po vertex shadri. Vstupom je jedno primitívum, v ktorom pre každý bod/hranu/trojuholník je shader prepočítavaný a výsledkom je skupina (alebo žiadne) grafických primitív (obr. 2.2).



Obrázok 2.2: Vytvorenie čajníka v každom vertexe originálneho modelu

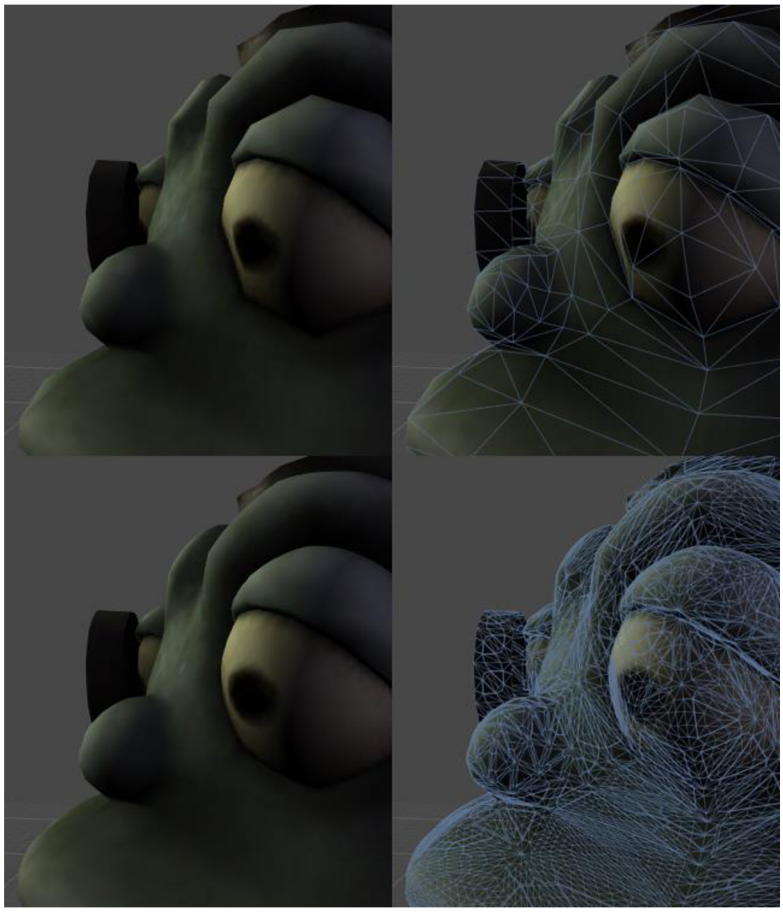
2.5.3 Tessellation Shader

Tento shader pridáva do klasickej grafickej pipeliny dve nové fázy: tessellation control a tessellation evaluation. Tieto dve fázy dohromady umožňujú jednoduchším meshom, aby boli za behu rozdelené menšie časti pomocou matematickej funkcie. Táto funkcia môže byť rôzne upravovaná podľa aktuálnych potrieb (napríklad vzdialenosť od kamery - obr. 2.3). Daným procesom je možné dosiahnuť efekt, pri ktorom sú objekty blízko kamery detailnejšie ako tie vzdialené.



Obrázok 2.3: Teselácia, založená na vzdialenosti od kamery

K teselácii môžeme pristupovať rôznymi spôsobmi. Môžeme na objekty aplikovať fixnú veľkosť teselácie - tento prístup je vhodný v prípade, že všetky/väčšina stien modelu má na obrazovke podobnú veľkosť. Ďalším prístupom je už spomínaná teselácia, založená na vzdialenosti objektu/meshu od kamery. Tento prístup je však vhodný predovšetkým vtedy, keď sú veľkosti trojuholníkov podobné. Riešením je prístup, založený na dĺžke hrán (tento prístup je vhodné kombinovať s prístupom, založeným na vzdialenosti od kamery). Posledným prístupom, ktorý je aj najvýhodnejší v pomere k dosiahnutým výsledkom, je Phongova teselácia (porovnanie založené na dokumente [6] v podkapitole 4.2.3.1.4, príklad na obrázku 2.4). V tomto prístupe čiastočne upravíme aj samotnú polohu rozdelených plôch, čím môžeme aj v low-poly meshoch dosiahnuť relatívne hladký povrch.

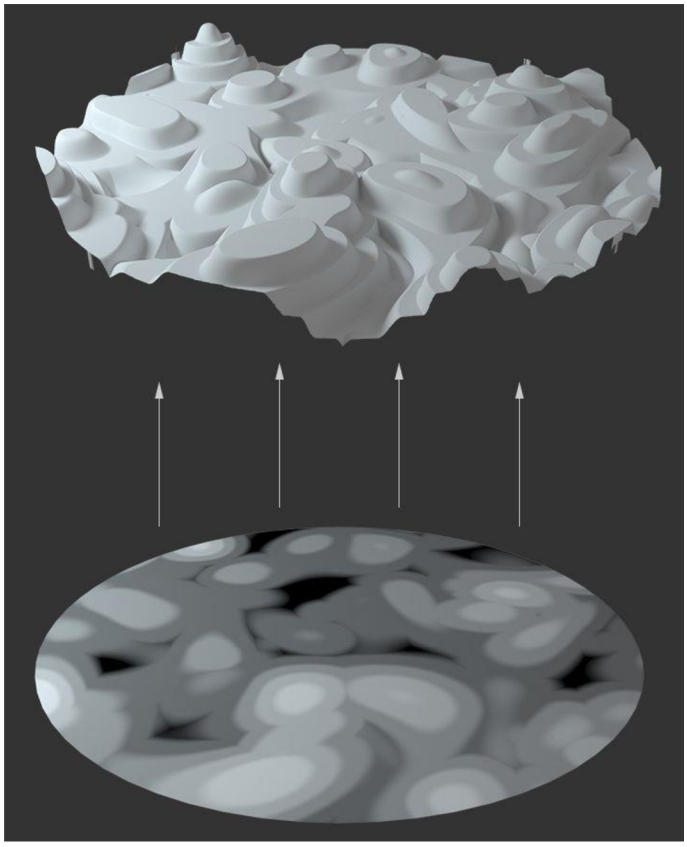


Obrázok 2.4: Porovnanie medzi klasickým (hore) shadrom a Phongovou teseláciou

2.6 Displacement mapping

Displacement mapping je technika, ktorá vytvára efekt, pri ktorom dochádza k skutočnému posunutiu vrcholov objektu (najčastejšie) pozdĺž lokálnych povrchových normálov, a to podľa hodnoty, ktorú vráti funkcia textúry (obr. 2.5). Táto hodnota je vypočítavaná pre každý jeden bod povrchu.

Nevýhodou tejto techniky je fakt, že je pomerne náročná. Na jej implementáciu sa používa textúra, kde jednotlivé texely reprezentujú silu displacementu v bode, na ktorý je táto textúra aplikovaná. V súčasnosti však mnoho rendererov poskytuje programovateľné shadovanie, ktoré dokáže vytvárať vysokokvalitné procedurálne textúry a v takom prípade je teda diskutabilné, či sa jedná o reálne mapovanie, keďže nedochádza k použitiu žiadnej reálnej textúry, ktorá by bola mapovaná, ale používa sa iba výsledok funkcie, generujúcej procedurálnu textúru.



Obrázok 2.5: Pokrivenie povrchu na základe displacement textúry

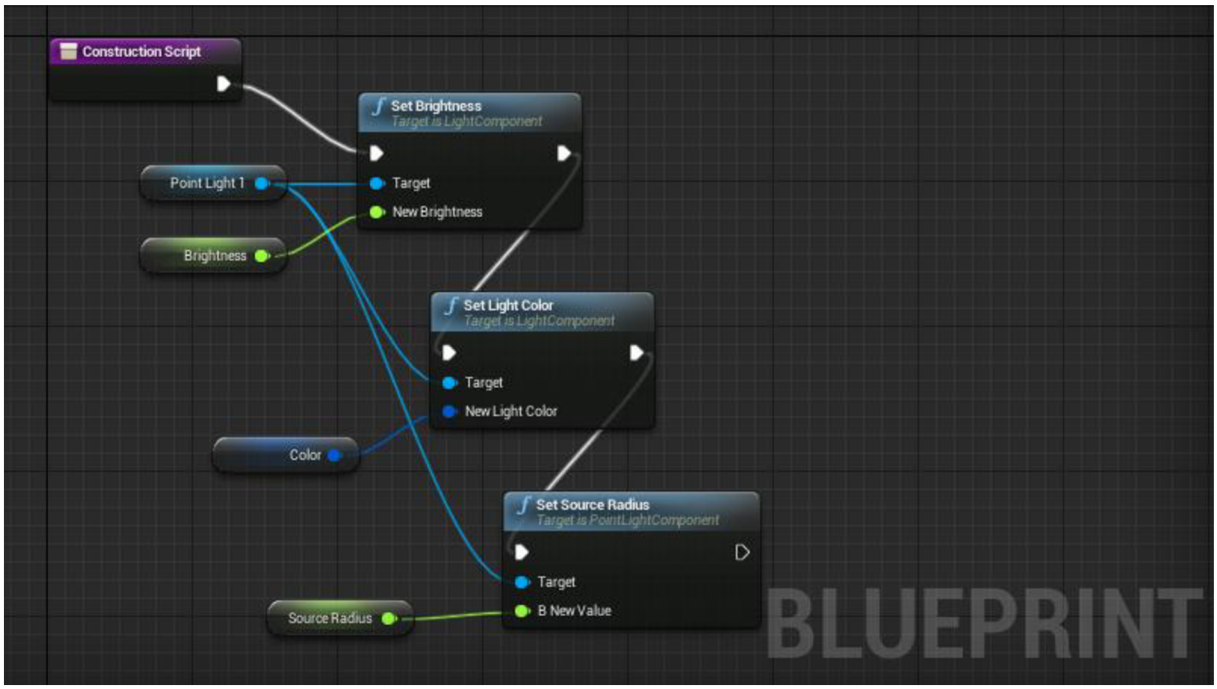
3 Popis Unity a návrh

V súčasnosti sú na trhu najpopulárnejšie a najčastejšie používané enginy Unity a Unreal Engine (respektíve UDK – unreal development kit). Ako som už spomenul, pre tento projekt som zvolil engine Unity a v tejto podkapitole by som opísal jeho výhody a nevýhody (hlavne v porovnaní s konkurenčným Unreal Enginom).

Unity vyniká v podpore vývoja pre mobily a v oblasti 2D hier, pričom dobrá podpora vývoja pre mobily je rozhodne plusom pre hry s jednoduchým ovládaním, ako je napríklad tento projekt (pričom sféra mobilných hier je v súvislosti s týmto projektom skôr cesta do budúcnosti ako aktívna súčasná snaha.)

Je pravda, že v oblasti 3D hier pre PC/konzoly je na tom Unreal lepšie (ponúka pokročilejšie nástroje na prácu s grafikou), ale pokiaľ nie je cieľom vývojára tvoriť hry s next-gen grafikou (čo nie je cieľom tohto projektu), tak je Unity rovnako dobrou voľbou.

Ďalším rozdielom je podpora jazykov. Unreal podporuje C++ a Unity zas C# a JavaScript, pričom popisy funkcionalít Unity vychádzajú z dokumentu [6] (Unity manuál). V zásade sa nejedná o veľký rozdiel, ale vzhľadom k tomu, že ja sa v súčasnosti zameriavam na objektovo orientované programovanie (OOP), ktoré má silnejšiu podporu v Unity, tak je pre mňa Unity jasnou voľbou. Je nutné dodať, že Unreal ponúka ešte nástroj Blueprint (obr. 3.1), ktorý dáva možnosť umožňuje vizuálne skriptovanie. Blueprint dáva možnosť jednoduchšie prototypovanie a testovanie levelov, no z osobnej skúsenosti môžem potvrdiť, že pri väčších projektoch sa výhody Blueprintu strácajú.



Obrázok 3.1: Ukážka skriptovania v Blueprinte

Čo sa týka jednoduchosti ovládania, tak sa mnohí užívatelia zhodujú (v súvislosti s projektom je to hlavne môj názor), že ovládanie Unity je intuitívnejšie a jednoduchšie na pochopenie, no Unreal prichádza so skutočne pokročilým UI, ktoré taktiež nie je práve najnáročnejšie na ovládanie.

3.1 Spracovanie zvuku v Unity

Unity poskytuje rôzne knižnice a objekty na prácu a spracovanie zvuku, pričom na účely tohto projektu sa používa konkrétne trieda AudioSource (vizualizácia na obrázku 3.2). Táto knižnica poskytuje funkciu GetSpectrumData, ktorá preberá ako parametre počet vzoriek, kanálov a orezávacie okno, pričom vráti blok, reprezentujúci dátové spektrum zdroja zvuku.

Nevýhodou tohto prístupu je, že vyžaduje prvok AudioClip, ktorý (v prípade, že sú piesne načítavané pri spustení hry a nie sú automaticky zabalené v hre) je nutné naplniť zvukovým súborom, ktorý sa načítava pomocou funkcie GetAudioClip triedy WWW, no v aktuálnom stave je pre Web/Standalone verziu dostupné len načítavanie formátov ogg a wav.

Obrázok 3.2: Objekt Audio Source z pohľadu editora

3.2 Podpora Shaderov v Unity

Unity podporuje tri druhy rôznych shadrov. Podporované druhy sú surface, vertex/fragment a fixed function shadre. S novšími verziami Unity pribudli aj ďalšie a s poslednou verziou Unity prišla aj podpora DirectX 11 a aj podpora tessellation shadrov.

Každý shader sa v Unity zabaľuje do ShaderLab kódu, vnútri ktorého sa potom nachádza samotný shader, ktorý je už napísaný buď v jazyku CG alebo HLSL, pričom fixed function shadre sa píšu zásadne v ShaderLabe. V prípade ostatných spôsobov sa v ShaderLab kóde vyznačí oblasť pre CG kód a v tejto oblasti sa potom bude výsledný shader nachádzať.

3.2.1 Image effects

Jednou z veľkých noviniek v Unity 5 oproti Unity 4 je podpora image (postprocessing) efektov už vo Free verzii, takže teraz sú dostupné všetkých užívateľom. Jedná sa o zvláštnu skupinu shadrov, ktoré

sú aplikované nie na materiál, ale na samotnú kameru. Snímky z hernej kamery sú interne prevádzané na materiál, ktorý je potom predaný shadru ako vstupný materiál, takže shader s ním dokáže pracovať rovnako ako s ktorýmkoľvek iným materiálom. Tento fakt nám umožňuje použiť všetky klasické shadre - ako napríklad vertex a fragment shadre - na výstup kamery.

Samotné Unity prichádza s celým balíkom shadrov, ktoré demonštrujú možnosti image efektov, no samozrejme, že je tu stále možnosť napísať si svoje vlastné. Pri písaní vlastných shadrov je však nutné myslieť na hlavný rozdiel medzi klasickými shadrami a image efektmi. Na implementovanie image efektu je potrebné okrem shadru napísať aj skript, ktorý implementuje ovládanie eventu OnRenderImage, v ktorom explicitne predá zachytený stav obrazovky ako materiál shadru, ktorý ho pred zobrazením na obrazovke ešte ďalej spracuje.

3.2.2 Surface Shader

Keďže písanie shadrov, ktoré pracujú s osvetlením, nie je ľahké, tak prišlo Unity s jednoduchým, no komplexným riešením, ktorým sú surface shadre. Surface shadre sa starajú o rôzne druhy svetiel, rôzne možnosti tieňov, rôzne druhy rendering path (forward a deferred), pričom toto všetko berie do úvahy tento druh shadrov.

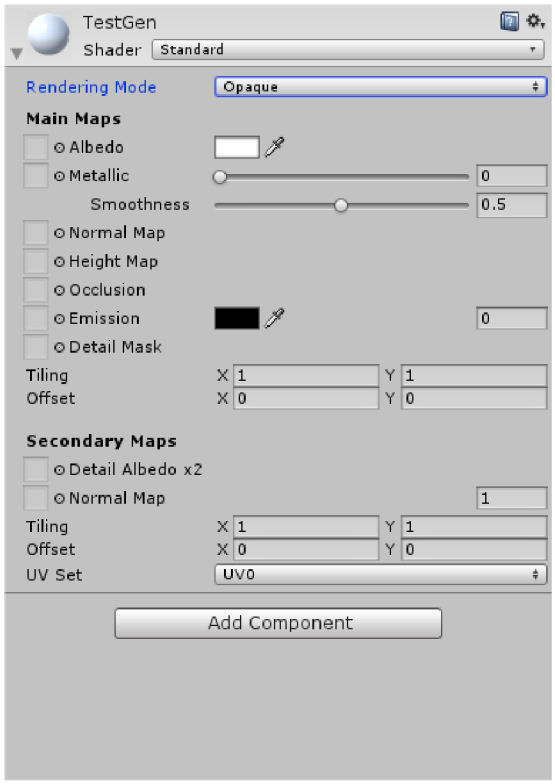
V skutočnosti sa jedná o prístup ku generovaniu kódu, a to je to, čo robí daný druh shadrov tak jednoduchým na používanie. Hlavnou úlohou je totiž generovanie často sa opakujúceho kódu, ktorý by inak bolo nutné napísať ručne. Zvyšok shadra sa píše rovnako ako ostatné shadre v Unity - v jazyku Cg/HLSL.

Základným princípom je nadefinovanie "surface funkcie", ktorá preberá texturovacie koordináty alebo iné potrebné dáta ako vstup, pričom počas behu tejto funkcie dôjde k naplneniu výstupnej *SurfaceOutput* štruktúry. Tá popisuje vlastnosti povrchu (preto Surface shader), ako sú napríklad albedo farbu, normály, svetelné emisie, spekularitu a iné.

Na základe vstupných informácií shader potom vygeneruje samotné vertex a pixel (fragment) shadre a aj jednotlivé priebehy, ktoré spracovávajú forward a deferred rendering.

Podmienkou použitia takéhoto shadra je definovanie osvetľovacieho modelu, ktorý má byť v shadri použitý. Unity v tomto smere podporuje ako physically based modely, tak aj jednoduché non-physically based modely (Lambert, BlinnPhong). Okrem už definovaných osvetľovacích modelov, je možné nadefinovať aj svoje vlastné.

Okrem týchto funkcionalít, surface shader umožňuje aj definovanie/použitie priehľadnosti a alpha testovania, nadefinovanie vlastnej vertex funkcie, spracovanie tieňov, teseláciu a mnoho ďalšieho (tento druh shadru bol vyvinutý s dôrazom na možnosť tvorby komplexných riešení, a preto existuje veľké množstvo rôznych voliteľných argumentov a prepínačov, ktorými je možné ovplyvňovať funkcionalitu shadru).



Obrázok 3.3: Vstavaná ukážka surface shadra

Ako je viditeľné na obrázku 3.3, tak je v editore štandardný surface shader označený jednoducho ako "Standard" (alebo "Standard (Specular setup)"), no samozrejme, sa jedná iba o ukážku možnosti tohto shadra - Unity bez problémov umožňuje naprogramovať si svoj vlastný.

4 Implementácia

Táto kapitola je venovaná samotnej implementácii projektu, pričom hlavným cieľom tejto kapitoly je vysvetliť problémy a riešenia vychádzajúce z prostredia vybraného pre tento projekt (eninge Unity).

4.1 Real-time načítavanie MP3

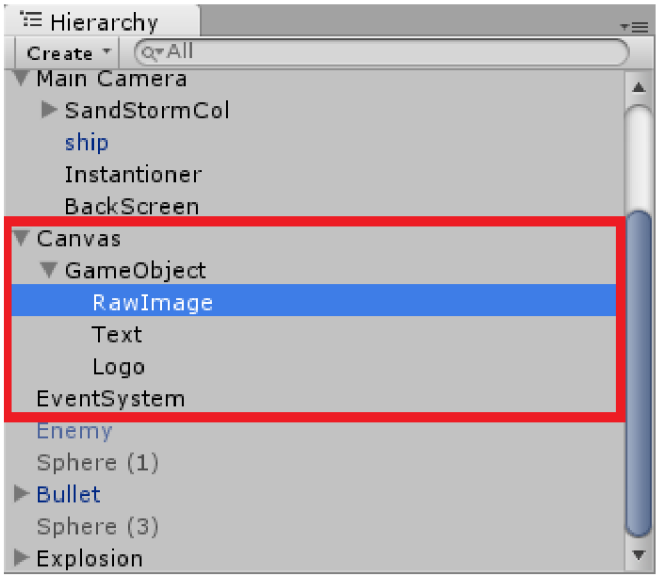
Základným problémom, ktorý dodnes v Unity pretrváva (z licenčných dôvodov), je podpora real-time načítavania MP3 súborov iba v builde pre mobily. Tento problém je však možné obísť tým, že sa človek nezameria výlučne na možnosti prostredia Unity, ale využije možnosti jazyka, ktorý je podporovaný Unity (C#).

Jedným z možných prístupov, je variant, ktorý využitý v tejto práci - spočíva v importovaní voľne dostupnej DLL knižnice MPG123, ktorú je možné následne použiť na vytvorenie knižnice, ktorá poskytne jednoduché rozhranie pre načítavanie MP3 súborov, keďže knižnica MPG123 poskytuje low-level funkcie pre prácu s MP3 súbormi, a to môže byť relatívne náročné na bežné používanie.

Celý proces je možné zhrnúť tak, že za behu aplikácie skrze skript načítame obsah MP3 súboru a tieto dáta potom vloží do premennej typu AudioClip (AudioClip je základná trieda enginu Unity, slúžiaca na prácu so zvukom) pomocou funkcie AudioClip.SetData.

4.2 Grafické komponenety a ich správanie

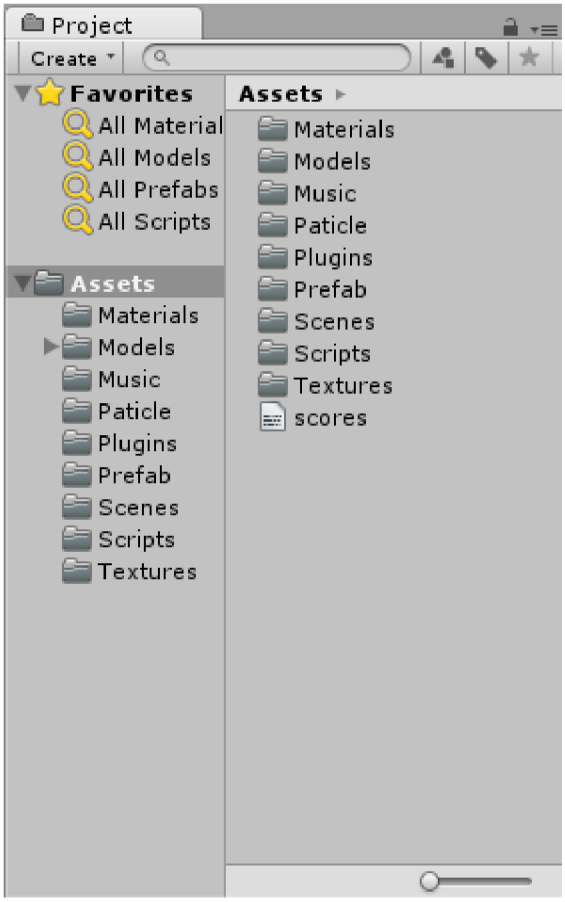
Od Unity 4.6 je možné v engine využívať nový GUI systém. Základom tohto systému je objekt typu Canvas. Vytvorenie akéhokoľvek GUI komponentu z prostredia Unity automaticky vytvorí aj tento objekt (a objekt typu EventSystem, ktorý slúži na obsluhovanie Eventov, vznikajúcich počas komunikácie medzi GUI a užívateľom). Pri vytváraní takéhoto systému je potrebné mať na pamäti, že poradie objektov (detí objektu Canvas) definuje, v akom poradí sú objekty vykresľované, pričom objekty sú vykresľované v poradí od najvyššieho po najnižší (obr 4.1).



Obrázok 4.1: Hierarchia "detí" objektu Canvas

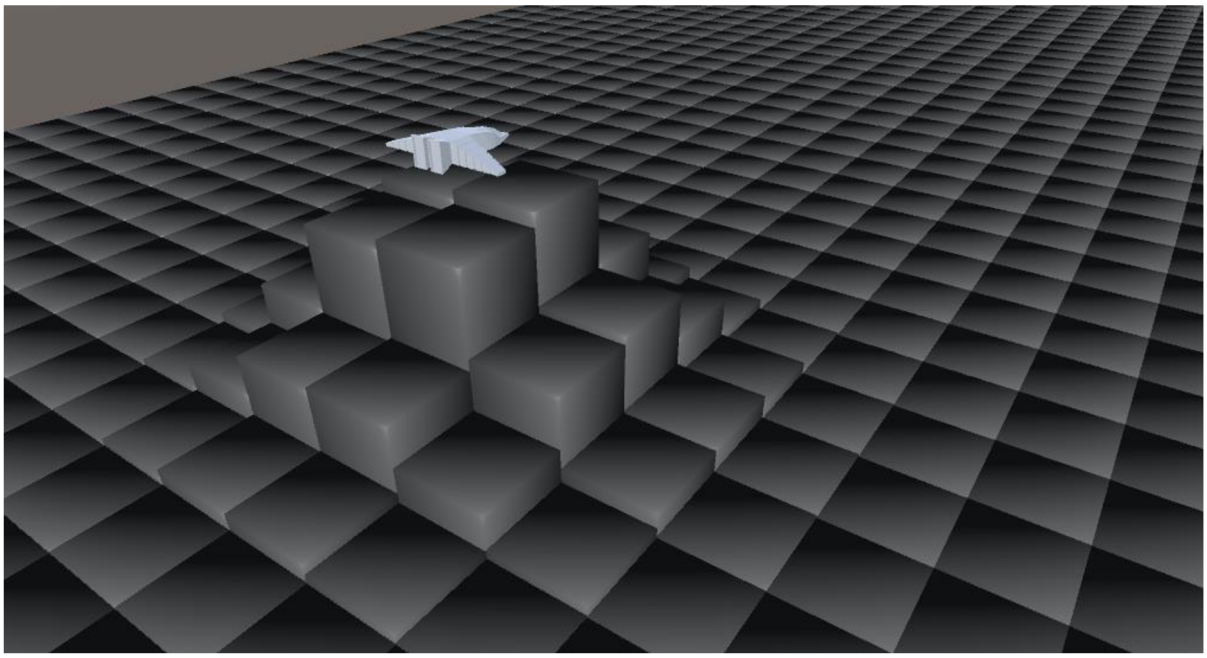
4.3 Vytvorenie scény

Všetky externé súčasti hry (modely, skripty, hudba, atď.) sa nachádzajú v zložke Assets v hlavom adresári projektu (obrázok 4.2). Pre vytvorenie scény potrebujeme dodať len model lode, textúry a hudbu, keďže ostatné súbory sú vytvárané z prostredia Unity a sú do tohto priečinka vkladané automaticky. Tento priečinok navyše plní podobnú úlohu ako dátové priečinky výsledného buildu projektu - interná premenná Application.dataPath totiž prechováva buď adresu zložky Assets v prípade editačného módu alebo adresu dátovej zložky v prípade hotového buildu.



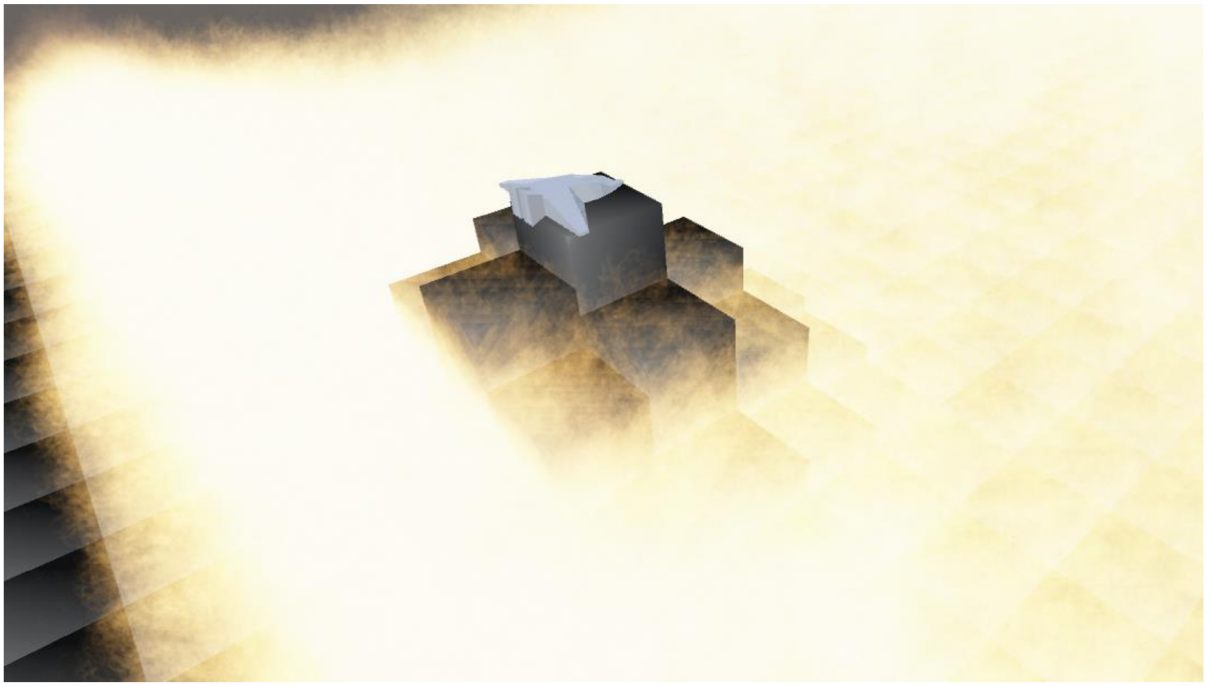
Obrázok 4.2: Zložka Assets a stromová štruktúra adresárov projektu

Samotná scéna obsahuje okrem lode interaktívnu "podlahu" (obrázok 4.3). Táto podlaha je vytváraná v priebehu hry. Interaktívna zložka spočíva v tom, že sa časť podlahy vysúva podľa polohy hráča v scéne. Najprv skrze skript je vytvorená obyčajná kockovaná sieť, na ktorú je potom aplikovaný geometry shader, ktorý na vstupe preberá trojuholníky a výstupom je kváder, ktorého poloha je upravená vzhľadom k vzdialenosti medzi ním a loďou. Vždy dva trojuholníky (jedno "oko" siete) sú potrebné na vytvorenie jedného kvádra. Tu je čiastočne vidieť nevýhodu geometry shadrov v Unity. Tento engin zatiaľ nepodporuje opakované volanie geometry shadra a preto, ak chceme dosiahnuť podobný efekt, tak musíme kombinovať spracovanie skrze skript a skrze shader, pričom musíme dbať na rozumné rozdelenie práce medzi procesor a grafickú kartu - na procesore vykonávať len čo najjednoduchšie úlohy, keďže vo veľkých scénach môže aj najmenšie navýšenie náročnosti spôsobiť veľký pokles v fps.



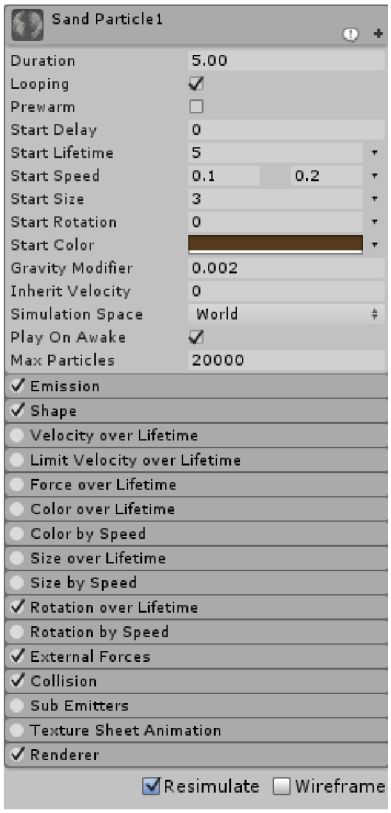
Obrázok 4.3: Interaktívna podlaha reagujúca na polohu hráča

Pre doplnenie vizuálneho efektu a zlepšenie príťažlivosti scény som využil Particle systém, ktorý Unity ponúka. Každý particle effect je vytváraný skrze tento systém, pričom Unity zároveň ponúka niekoľko rôznych variantov podľa aktuálnej potreby užívateľa - bodový zdroj, eliptický zdroj, meshový alebo "celosvetový" zdroj.



Obrázok 4.4: Slnečná búrka vytvorená cez particle systém

K vytvoreniu efektu slnečnej búrky som využil bodový zdroj, ktorý po nastavení vhodnej kombinácie hodnôt vytvoril výsledok zobrazený na obrázku 4.4. Ďalším dôležitým prvkom, potrebným k vytvoreniu tohto efektu je pridanie Wind Zone komponenty kdekoľvek do scény. Tento komponent vytvára na mieste svojej existencie simuláciu vetra. Vďaka simulovanému vetru je jednoduchšie vytvoriť efekt slnečnej búrky a tá dáva programátorovi väčšiu kontrolu nad správaním sa jednotlivých častíc, keďže particle systém neponúka detailnú kontrolu nad správaním sa jednotlivých častíc (príklad na obrázku 4.5).

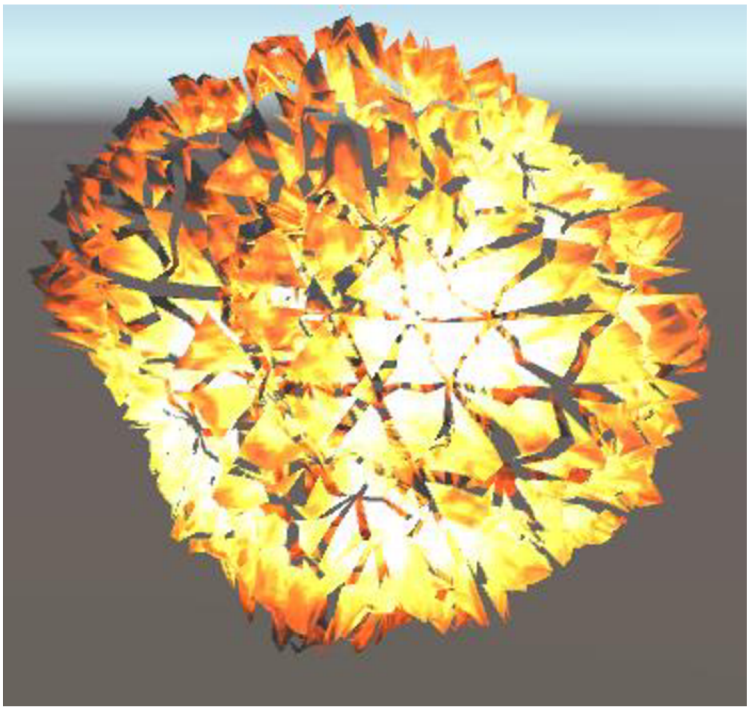


Obrázok 4.5: Particle system inspector

Skrze particle effects boli navrhnuté aj "laserové strely", s pomocou ktorých sa hráč bráni voči prilietajúcim asteroidom. Práca s particle effects je založená na vhodnej textúre jednej častice a vhodným nastavením jej emitorov a rendererov. Výsledný laserový lúč je možné vidieť na obrázku 4.6. Tento efekt by bolo možné vylepšiť animovaním a to tak, že by jednotlivé častice boli neustále emitované, pričom ich životnosť bola obmedzená, takže by vytvorili efekt lúča. Negatívnym efektom tohto variantu je fakt, že len ťažko sa kontroluje presné smerovanie častíc, takže výsledný efekt nepôsobil ani ako lúč (alebo strela), ale skôr ako náhodný blesk/výboj. Aj preto som sa rozhodol v hre ponechať pôvodnú verziu môjho riešenia.

Obrázok 4.6: Laserová strela

V neposlednom rade je otázkou interakcia strely a asteroidu. Najjednoduchším riešením je oba objekty jednoducho nechať zmiznúť, no tento efekt nič nepridáva výslednému vizuálnemu efektu hry, skôr naopak. Lepším riešením je využiť vertex shader s displacement mappingom, pričom toto riešenie aplikujeme na sférický objekt. Na takejto sfére potom môžeme simulovať efekt explózie, pričom daný efekt je možné vylepšiť pridaním menšej sféry do vnútra väčšej, čím sa zamedzí nepríjemnému efektu, kedy je jadro explózie viditeľne prázdne. Výsledný efekt tohto postupu je vidieť na obrázku 4.7.

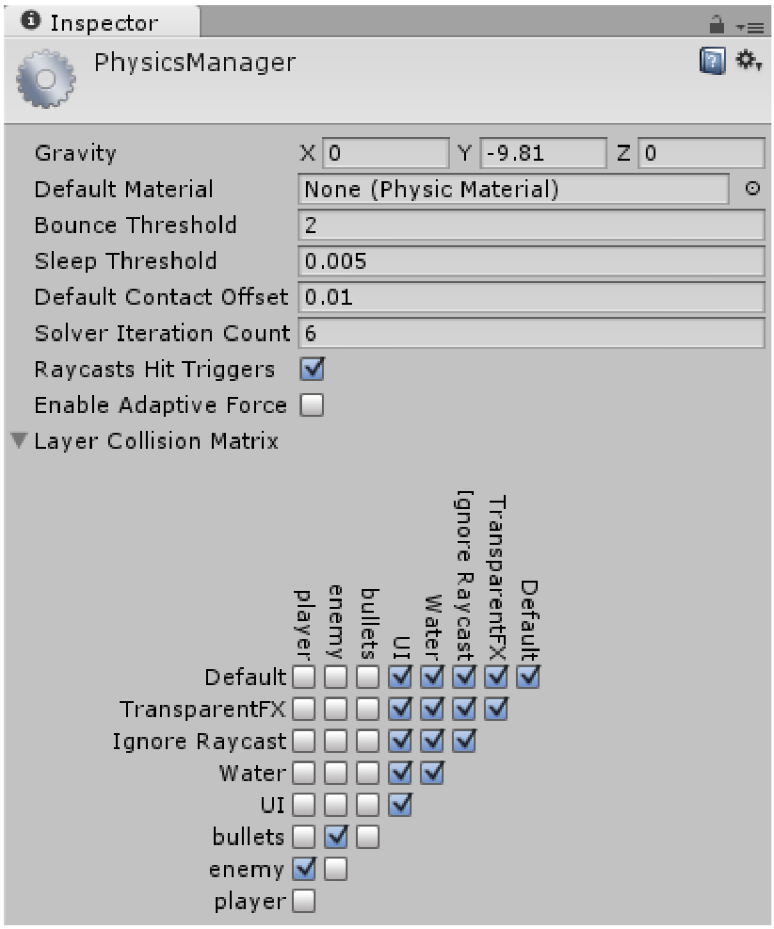


Obrázok 4.7: Simulovaná explózia

4.4 Fyzikálna interakcia

Keďže počas priebehu hry je nový objekt generovaný pre každý detekovaný "beat", tak je tu vysoká pravdepodobnosť, že v jeden okamih bude v scéne veľké množstvo objektov. Unity sa síce snaží znížiť záťaž procesora a grafickej karty tým, že niektoré výpočty spája, aby nemuseli byť vykonávané opakovane pre každý objekt (proces zvaný batching - sa využíva pri texturovaní). Fyzikálna interakcia je však unikátna pre každý objekt a keďže potrebujem, aby k nejakým interakciám dochádzalo, tak to znamená vysokú výpočtovú záťaž.

Túto záťaž je však možné znížiť, a to s využitím Layer-Based Collision Detection. Jedná sa o detekciu kolízií, založenú na "vrstvách". Každý objekt v scéne je možné priradiť k určitej vrstve a potom definovať, ktoré vrstvy môžu vzájomne jedna na druhú reagovať. Toto spôsobí, že niektoré interakcie sú jednoducho ignorované. V prípade malého množstva objektov nie je rozdiel veľmi citeľný, no v našom prípade má výrazný vplyv na fps hry.



Obrázok 4.8: Physics Manager spolu s maticou interakcií

Na obrázku 4.8 je zobrazený PhysicsManager (karta so základnými nastaveniami fyziky pre daný Unity projekt) spolu s maticou interakcií medzi jednotlivými vrstvami. Ako príklad uvediem vzťah bullets/bullets (alebo enemy/enemy), ktorý je odčiarknutý. Tým som odstránil fyzikálnu interakciu medzi objektami, ktoré patria pod tú istú vrstvu (v tomto prípade vrstvy bullets a enemy). Na prvý pohľad sa môže zdať, že by bolo realistickejšie toto políčko nechať zaškrtnuté, keďže aspoň v prípade enemy by asteroidy, (ktoré patria pod túto vrstvu), mali byť schopné narážať jeden do druhého, no prax ukázala, že spôsob instancovania týchto objektov - jeden spawner, z ktorého sú vypúšťané - znamená, že tieto objekty reagujú jeden s druhým už v momente vzniku, čo má za následok hromadenie objektov v mieste ich vzniku a kvôli rýchlo rastúcemu množstvu interakcií rýchlo narastá aj záťaž na výkon, a teda klesá fps.

Z týchto dôvodov je oveľa výhodnejšie ísť s menej realistickým, no výkonovo menej náročným riešením, kedy objekty tej samej vrstvy nemôžu (po stránke fyziky) reagovať na objekty tej istej vrstvy.

Na obrázku 4.8 je možné vidieť začiarknuté políčka pre ďalšie vrstvy ako Default, TransparentFX a pod. No v týchto prípadoch išlo buď o interakcie, ku ktorým v tomto projekte nedochádzalo, alebo išlo o interakcie, ku ktorým nedochádzalo často, a teda bolo možné si ich dovoliť nechať aktivované.

4.5 Doplnkové herné prvky

Jedným z dôležitých herných prvkov je práve GUI, ktoré bolo spomínané v podkapitole 4.2. Pre potreby tohto projektu a vzhľadom na aktuálny rozsah projektu nebolo nutné vytvárať zbytočne komplikované grafické užívateľské rozhranie. Medzi zásady dobrého GUI totiž patrí jednoduchosť a dobrá čitateľnosť. V niektorých prípadoch je síce vhodné vytvárať zložitý systém GUI, no to je vhodné len v prípade, kedy samotná hra využíva komplexné herné mechanizmy, alebo obsahuje iné komplexné/komplikované prvky, ktoré je nutné vysvetliť alebo nastaviť.

Obrázok 4.9: Úvodná obrazovka videná v editačnom režime

Zjednodušene sa dá povedať, že zložitosť (a celková komplexnosť GUI) by nemala prekračovať zložitosť hry a mechaník, s ktorými sa hráč počas hry stretáva (interné mechanizmy sú pred hráčom skryté, a teda ich komplexnosť nemá vplyv na potreby komplexnejšieho GUI).

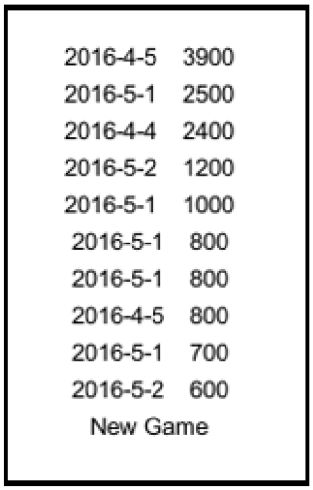
Na obrázku 4.9 je vidno úvodnú obrazovku, ktorá obsahuje logo hry a zoznam rozpoznaných pesničiek, z ktorých si hráč môže vybrať. Tento obrázok však obsahuje ešte jednu zaujímavosť, ktorá nie je súčasťou buildu, ale je viditeľná iba v editačnom režime - Resource Manager, ktorý sa nachádza v pravom hornom rohu. Toto okno ukazuje aktuálne vyťaženie procesora a grafickej karty (a ďalšie informácie vhodné pre ladenie). Resource Manager však už nespadá pod túto kapitolu a detaily jeho fungovania a používania spadajú už mimo rozsah tejto práce.

4.6 Prechovávanie dát medzi spusteniami

Jediné dáta, ktoré sú v prípade tohto projektu nutné prechovávať medzi rôznymi spusteniami je skóre, priradené ku konkrétnej piesni. Kvôli možnosti prechovávania dát nielen medzi jednotlivými spusteniami hry, ale aj v prípade presunu hry na iný počítač, rozhodol som sa dáta prechovávať cez najzákladnejší (nie nutne najjednoduchší) spôsob - skrze externý súbor, uchovávajúci ako dosiahnuté skóre, tak aj názov pesničky, ku ktorej je dané skóre priradené.

Hra na začiatku skontroluje existenciu súboru, obsahujúceho skóre a pripraví si výsledkovú listinu, obsahujúcu defaultné hodnoty, pozostávajúcu z dvojíc ??? a 0, kde ??? stoja na mieste dátumu a 0 je základná hodnota skóre. Potom program skontroluje existenciu výsledkového záznamu v súbore pre aktuálne zvolenú pesničku, a ak existuje, tak z neho načíta dáta do internej výsledkovej listiny. Táto výsledková listina je na konci hry updatnutá o novú hodnotu, ktorá je vložená na správne miesto (tento krok nie je ovplyvnený tým, či v minulom kroku bol alebo nebol nájdený existujúci záznam). Hneď po vykonaní tohto updatu je aktuálna interná tabuľka zapísaná na miesto v súbore, kde sa nachádzala pôvodná tabuľka pre aktuálne hranú pesničku.

Dáta sú zapisované do obyčajného textového súboru a hra nekoná kontrolu správnosti dát, čo je možným miestom pre budúce zmeny. Tento prístup v aktuálnom stave a pre aktuálne potreby je dostačujúci, lebo žiadne aktuálne možné interakcie s hrou buď nemôžu spôsobiť korupciu súboru s výsledkovou listinou alebo sa nachádzajú mimo interakcie, potrebné pre tento projekt a je teda možné zatiaľ vylúčiť ich vplyv. Samozrejme, je jasné, že v prípade pokračovania prác na tomto projekte je nutné tieto dáta buď šifrovať, ukladať ako binárny súbor alebo iným spôsobom znemožniť ich čítanie mimo herné prostredie. Ďalším podstatným krokom by bola kontrola obsahu súboru s výsledkovou listinou a zabezpečenie správnosti dát.



Obrázok 4.10: Jednoduchá výsledková listina na konci levelu

Ako je možné vidieť na obrázku 4.10, tak v aktuálnom stave hra zobrazuje len jednoduchú výsledkovú listinu, pozostávajúcu zo skóre a dňa, kedy bolo toto skóre dosiahnuté. V hre nie je nutné zobrazovať názov pesničky, pre ktorú bolo skóre dosiahnuté, keďže hráč si pesničku vyberal už na začiatku, takže vie, ku ktorej pesničke je toto skóre viazané.

4.7 CRT effect

Jedná sa o vizuálny efekt starého televízora (priblíženie tohto postupu je popísané na stránke [7]). Tento efekt je možné dosiahnuť s použitím image efektov zo štandardnej knižnice efektov. Základ tvoria dva shadre: Noise and Grain (obrázok 4.11) a Vignette and Chromatic Aberration shader (obrázok 4.12).

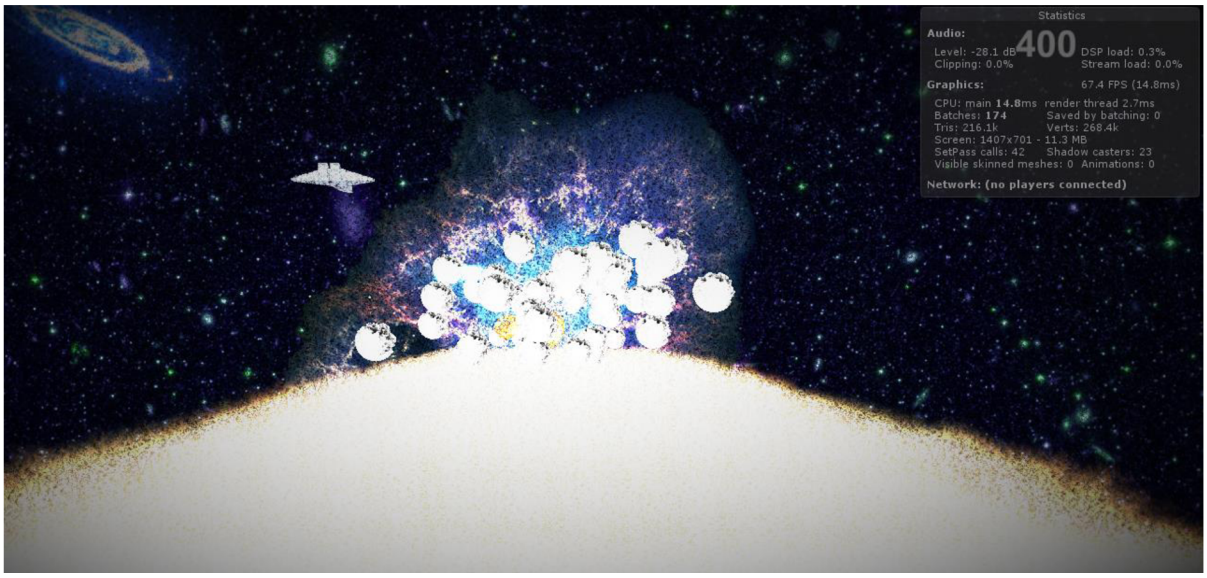
Noise and Grain shader sa postará o simuláciu známeho zrnenia obrazovky, kým Vitgnette and Chromatic Aberration sa postarajú o efekt "vypálenia" obrazovky, kedy stred je jasnejší ako okraje.

Obrázok 4.11: Noise and Grain shader

Obrázok 4.12: Vignette and Chromatic Aberration shader

4.8 Výsledný produkt

Po poskladaní scény, nastavení GUI a ostatných ovládacích a herných prvkov dostaneme výsledný produkt, ktorý je zobrazený na obrázku 4.13.



Obrázok 4.13: Herná obrazovka

Na prvý pohľad sa môže zdať, že nie je práve najprehľadnejšia, no to je súčasťou dizajnu a chceného efektu. Farby v okolí stredu obrazovky splývajú, kým k okrajom sa farby strácajú do čierna. Výsledkom je lacné zvýšenie náročnosti, no toto je presne taký typ náročnosti, ktorý sa používal v prvopočiatkoch herného priemyslu, kedy kvôli nedostatku pamäte developeri nemali inú šancu ako zneužiť niektorý už implementovaný herný prvok alebo mechanizmus k navýšeniu celkovej obtiažnosti. Takto téma, herné mechanizmy a aj vizuálny vzhľad hry spolu ladia.

5 Záver

V tomto projekte som sa zameral na demonštráciu možností enginu Unity pri vytváraní hry, využívajúcej analýzu hudby a procedurálne generovanie. V projekte som sa snažil využiť čo najširšiu škálu poskytovaných nástrojov, ako sú shadre, vstavané knižnice, podpora jazykov, fyzikálne simulácie a iné. Unity je engin so širokou škálou použití a aj začiatočníkovi poskytuje pokročilé nástroje, ktoré nie je ťažké použiť v projekte (ako bolo demonštrované práve pri shadroch a particle systémoch). Cieľ projektu bol naplnený, no stále je tu čo zlepšovať.

Každý použitý efekt je možné rozšíriť. Efekt starej CRT obrazovky je možné rozšíriť o rôznofarebné preskakujúce pruhy. Samotná "hviezdna cesta", po ktorej hráč letí, sa môže vplyvom hudby točiť. Toto by mohlo byť dosiahnuté tak, že geometry shader, vytvárajúci túto plochu, by bol doplnený o ďalšiu premennú reprezentujúcu bod, ku ktorému sa má trasa zatáčať. Takisto parametre, získané z hudby je možné použiť na kompletnú deformáciu tvaru asteroidu alebo na generovanie textúry, ktorá by presne zodpovedala aktuálnej pesničke. Celkovo je možné projekt doplniť o efekty, ktoré by dokonale zladili vizuálnu stránku hry s práve hrajúcou hudbou.

Literatúra

[1] Ward, Jeff: What is a Game Engine? [online]. [cit. 2016-05-11]. Dostupné na URL: http://www.gamecareerguide.com/features/529/what_is_a_game_.php

[2] Gustavson, Stefan.: Simplex noise demystified [online]. [cit. 2016-05-11]. Dostupné na URL: http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

[3] Arndt, Jörg: FXT: a library of algorithms [online]. [cit. 2016-05-11]. Dostupné na URL: http://www.jjj.de/fxt/

[4] Mallat, Stéphane. *A Wavelet Tour of Signal Processing: The Sparse Way*. With contributions from Gabriel Peyré. 3. vyd. [s.l.] : Academic Press, c2009. xx, 805 s. ISBN 978-0-123-74370-1

[5] Patin, Frédéric: Beat Detection Algorithms [online]. [cit. 2016-05-11]. Dostupné na URL: http://archive.gamedev.net/archive/reference/programming/features/beatdetection/index.html

[6] Unity, kolekcia autorov: Unity Manual [online]. [cit. 2016-05-11]. Dostupné na URL: http://docs.unity3d.com/Manual/index.html

[7] Zucconi, Alan: Screen shaders and image effects in Unity3D [online]. [cit. 2016-05-11]. Dostupné na URL: http://www.alanzucconi.com/2015/07/08/screen-shaders-and-postprocessing-effects-in-unity3d/

Zoznam príloh

- Technická správa vo formáte .docx
- Technická správa vo formáte .pdf
- Dokumentácia vo formáte .txt
- Projektové súbory Unity
- Spustiteľný standalone build hry