UNIVERZITA SV. CYRILA A METODA V TRNAVE
FAKULTA PRÍRODNÝCH VIED
KATEDRA APLIKOVANEJ INFORMATIKY

VÝVOJ 2D PC HRY - ARKANOID V UNITY

NÁVODY NA CVIČENIA

JANA JURINOVÁ

UNIVERZITA SV. CYRILA A METODA V TRNAVE

FAKULTA PRÍRODNÝCH VIED

Katedra aplikovanej informatiky

VÝVOJ 2D PC HRY – ARKANOID V UNITY

Návody na cvičenia

Jana Jurinová

Trnava, 2022

Autor:

Ing. Jana Jurinová, PhD.

Recenzenti:

Ing. Miroslav Beňo, PhD.

PaedDr. Patrik Voštinár, PhD.

Skriptá boli schválené Edičnou radou Univerzity sv. Cyrila a Metoda v Trnave a vedením Fakulty prírodných vied UCM.

Vydané s podporou grantu KEGA 017UCM-4/2022 „Vývoj interaktívneho e-kurzu s využitím „SMART“ technológií na rozvoj algoritmického myslenia a programátorských zručností“.

Za odbornú a jazykovú stránku týchto skrípt zodpovedá autorka.

Rukopis neprešiel redakčnou ani jazykovou úpravou.

Vydavateľ: Univerzita sv. Cyrila a Metoda v Trnave, 2022

1. vydanie

ISBN 978-80-572-0225-7

Predhovor

Tieto skriptá sú určené pre študentov študijného odboru Aplikovaná informatika a pre všetkých, ktorí majú záujem rozvíjať svoje vedomosti v oblasti vývoja a programovania 2D hier pomocou herného enginu Unity.

Problematika spracovaná v tejto publikácii predpokladá vedomosti čitateľa z oblasti objektovo orientovaného programovania, algoritmického riešenia problémov, ako aj grafického spracovania. Snahou je predstaviť problematiku vývoja 2D hry v Unity na konkrétnom type hry, pričom skriptá povedú čitateľa krok za krokom celým vývojom tejto hry – od návrhu až po export. Je teda zrejmé, že tieto skriptá nepokrývajú celú komplexnú problematiku vývoja hier a ani všetky možností herného enginu Unity.

Štruktúra a radenie jednotlivých kapitol vyplýva z logiky budovania herného mechanizmu hry. Tieto skriptá vychádzajú z oficiálnej dokumentácie k Unity. Obsah je doplnený o ilustrácie a obrázky zachytávajúce nastavenia priamo v editore Unity, alebo vo Visual Studio s cieľom podporiť vnímanie a chápanie praktickej aplikácie. Hra bola vyvíjaná v Unity vo verzii 2020.1.1f1. Kapitoly sú doplnené o kontrolné otázky a úlohy na overenie vedomostí čitateľa a motiváciu na ďalšie rozširovanie hry. Na trhu existujú desiatky, ak nie stovky publikácií venujúcich sa vývoju hier a objasňovaniu konceptov v Unity. Zoznam použitej a odporúčanej literatúry slúži pre čitateľov ako potenciálny zdroj na rozšírenie poznatkov o skúmanej problematike.

Skriptá vychádzajú v elektronickej podobe na základe dvoch faktorov. Grafická reprezentácia niektorých ilustrácií ani pri použitom formáte A4 nie je dostatočne čitateľná a vyžaduje možnosť zväčšenia. Druhým dôvodom je snaha uľahčiť čitateľom prácu, preto súčasťou skrípt sú zdrojové kódy a súbory riešeného projektu, vďaka ktorým si môžu okamžite overiť ich funkčnosť a priamo s nimi pracovať. V skriptách sa tiež nachádzajú hypertextové odkazy, ktoré odkazujú priamo na dokumentáciu Unity s cieľom poskytnúť komplexné informácie.

Terminológia v informačných technológiách a informatike z veľkej časti pochádza z angličtiny a používatelia využívajú originálne anglické výrazy. Angličtina sa stala odborným žargónom IT špecialistov. Pre vývojára akéhokoľvek softvéru je znalosť anglického jazyka prakticky nevyhnutná, aspoň na úrovni čítania a porozumenia dokumentácií. Na viacerých miestach používame anglickú terminológiu kvôli lepšiemu pochopeniu a z dôvodu, že niektoré preklady sú skôr zmätočné, než prínosné.

Ak by ste objavili nejaké chyby, alebo by vám nejaké informácie v týchto skriptách chýbali, neváhajte ma kontaktovať na jana.jurinova@ucm.sk. Privítam akékoľvek pripomienky a námety na vylepšenie obsahu.

Rada by som na tomto mieste poďakovala recenzentom Ing. Miroslavovi Beňovi, PhD. a PaedDr. Patrikovi Voštinárovi, PhD. za ich prínosné odborné komentáre, poznámky a námety, ktoré prispeli k skvalitneniu týchto skrípt.

Január 2022 Ing. Jana Jurinová, PhD., autorka učebnice

Obsah

1 Čo je Arkanoid?

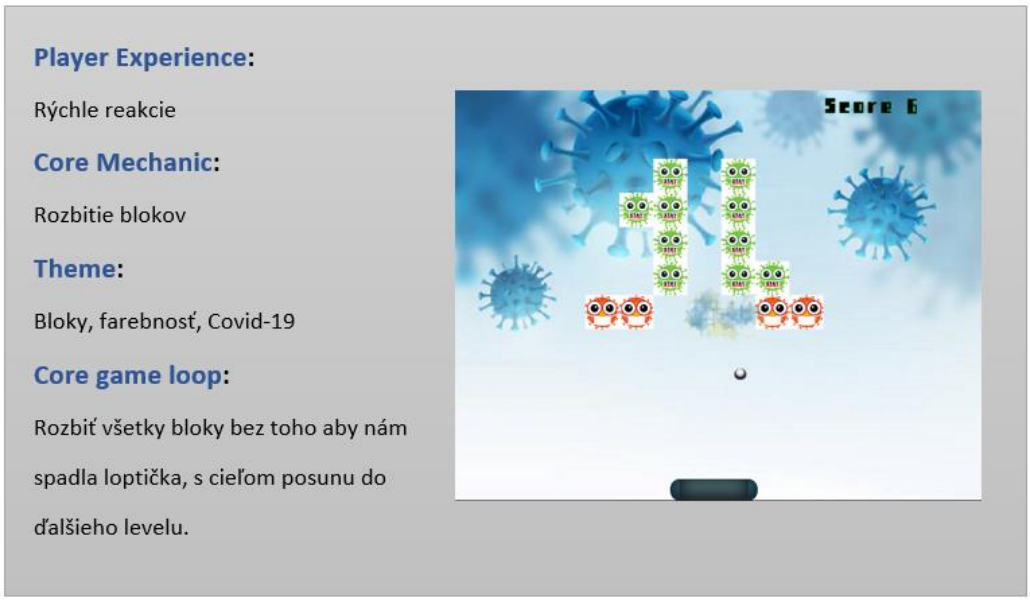
Hru Arkanoid môžeme považovať už za legendu vzhľadom na to, že jej historický vývoj siaha až do roku 1986. Napriek tomu túto hru pozná aj dnešná mladá generácia a je inšpiráciou aj pre vývojárov 21. storočia. Ako uvádza Zelenka (2012) môžu za to jednoduché herné mechanizmy a hrateľnosť, ktorá spôsobuje, že hraním hry hráč strávi aj niekoľko hodín dňa. Túto hru môžeme kategorizovať ako „*Block Breaker Arcade Game*“. Základný princíp hry spočíva v ovládaní plošiny (padla), pomocou ktorého odrážame loptičku a tá demoluje stenu tvorenú rôznymi geometrickými objektmi = blokmi. V prípade, že sa loptičku nepodarí odraziť, vo väčšine prípadov „strácate život“, alebo hra končí. Od jej vzniku bolo vytvorených viacero jej kópií a modifikácií, ktoré sa vo väčšine prípadov líšia vizuálnou stránkou, pridaním nejakého herného príbehu, ale aj doplnením rôznych bonusov, ktoré padajú z demolovaných geometrických objektov predstavujúcich stenu v priebehu hry a prinášajú rôzne vylepšenia (napr. viacero loptičiek, väčšie padlo a pod.), ale aj zhoršenia (napr. menšie padlo, ovplyvnenie rýchlosti loptičky, skrátenie času levelu a pod.). Rôzne správanie vykazujú aj bloky, ktoré môžu byť nezničiteľné, alebo je nutné ich trafiť viackrát. Niektoré modifikácie hry sú doplnené napríklad o rôzne módy hry. Cieľom je napríklad nazbierať čo najvyššie skóre, alebo je každý level podmienený časovačom, prípadne existujú aj také verzie, ktoré podporujú hru viacerých hráčov ovládaných z jedného počítača. Viac o tomto type hry píše Nelson (2021). Aj naša verzia stavia na týchto prvkoch, pričom vizuálna reprezentácia hry je veľmi úzko spojená so spoločensky dominantnou témou pandémie Covid-19. Grafické prevedenie hry však nie je primárnym aspektom týchto skrípt.

V súvislosti s Unity budeme pri vývoji tejto hry pracovať s fyzikou. Použijeme *Unity physics engine*, budeme simulovať gravitáciu, budeme riešiť kolízie objektov = *Collision* a s tým súvisiace *Rigidbody*. Tiež využijeme *Particle Effect* na pridanie jednoduchej animácie a pridáme aj nejaké zvukové efekty.

1.1 Základný návrh hry = *Game Design*

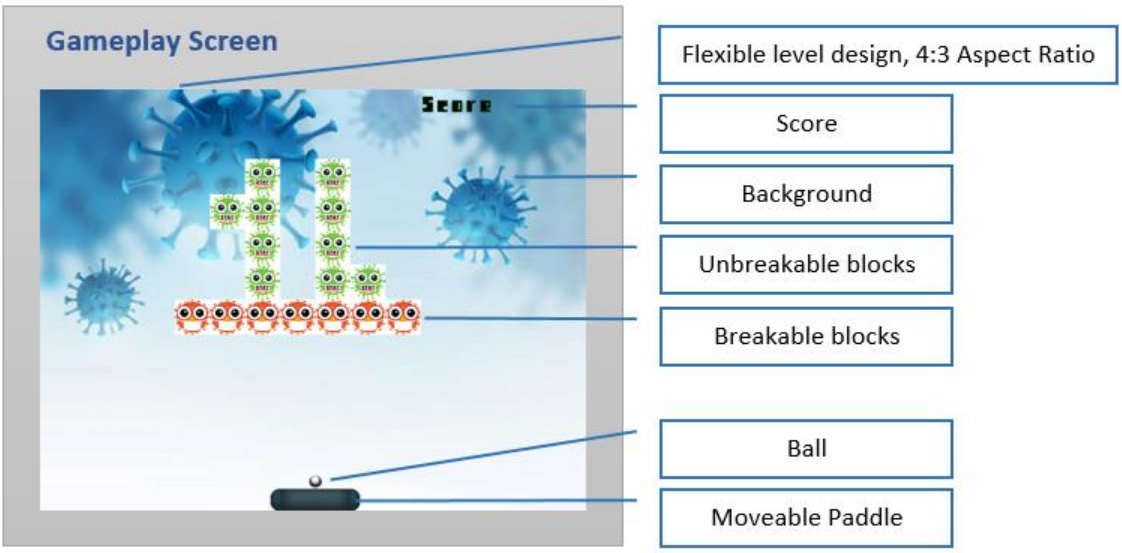
Základným stavebným prvkom pri každom vývoji hry je vytvorenie *Game Designu* pre vyvíjanú hru. Jeho podstatou je zachytiť návrh hry od prvotnej myšlienky cez vytvorenie témy, príbehu a pravidiel až po detailný princíp hry.

Ako prvé sa pokúsime zachytiť základný cieľ a princíp hry, ktorý je súčasťou *Game designu*:



Obr. 1 *Game Design* hry.

Obrázok 2 predstavuje základné stavebné prvky hry:



Obr. 2 Základné stavebné prvky hry.

Pomer 4:3 dáva dostatočný priestor na to, aby sa loptička príliš rýchlo neodrážala a hra bola hrateľná. Ak by bola hra plytšia, t. j. nižšia vo vertikálnom smere, tak pri náraze by sme mali veľmi rýchlu reakciu loptičky, a pri jej lete v sklone zase pomalú.

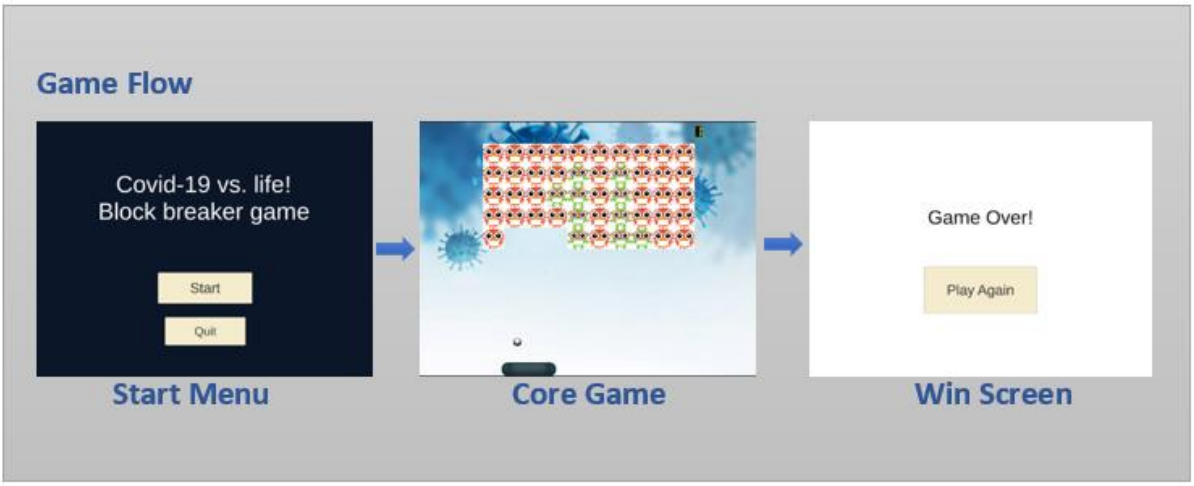
1.2 Kontrolné otázky a úlohy

1. Čo vyjadruje termín *Game Design*?
2. Premyslite si tému vašej hry, samozrejme môžete kopírovať moju.
3. Nájdite si vhodné obrázky pre základné stavebné prvky hry.
4. Premyslite si názov hry.

Tip autora: Bez ohľadu na použité grafické prvky, je vhodné dodržiavať pomenovania uvedené v skriptách z dôvodu, že tieto názvy budeme často používať v kódoch a pomocou nich sa na ne budeme odkazovať.

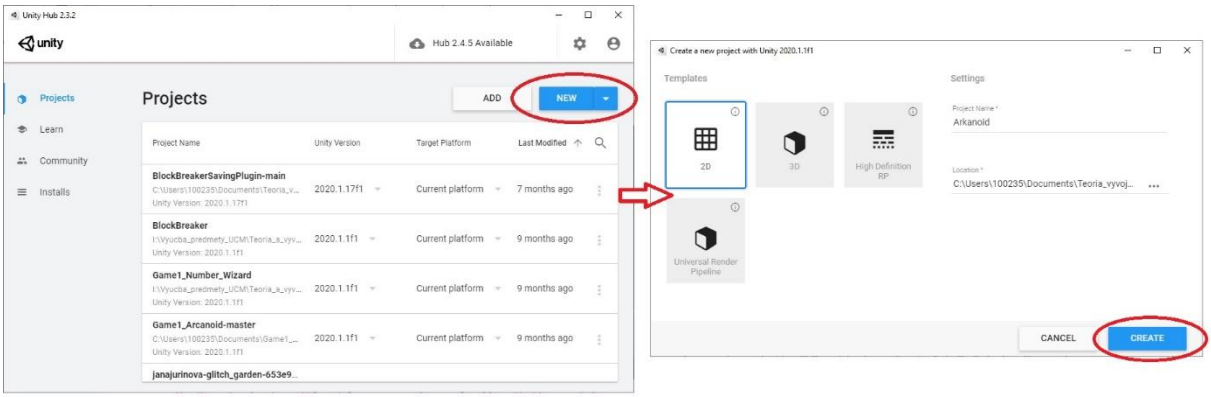
2 Vytvorenie projektu

Hlavný cieľom tejto kapitoly je naučiť sa vytvoriť projekt v Unity a zabezpečiť prechod cez jednotlivé scény, ktoré budú predstavovať úvodnú obrazovku, jednotlivé levely ako aj záverečnú obrazovku s vyhodnotením (Obr. 3).



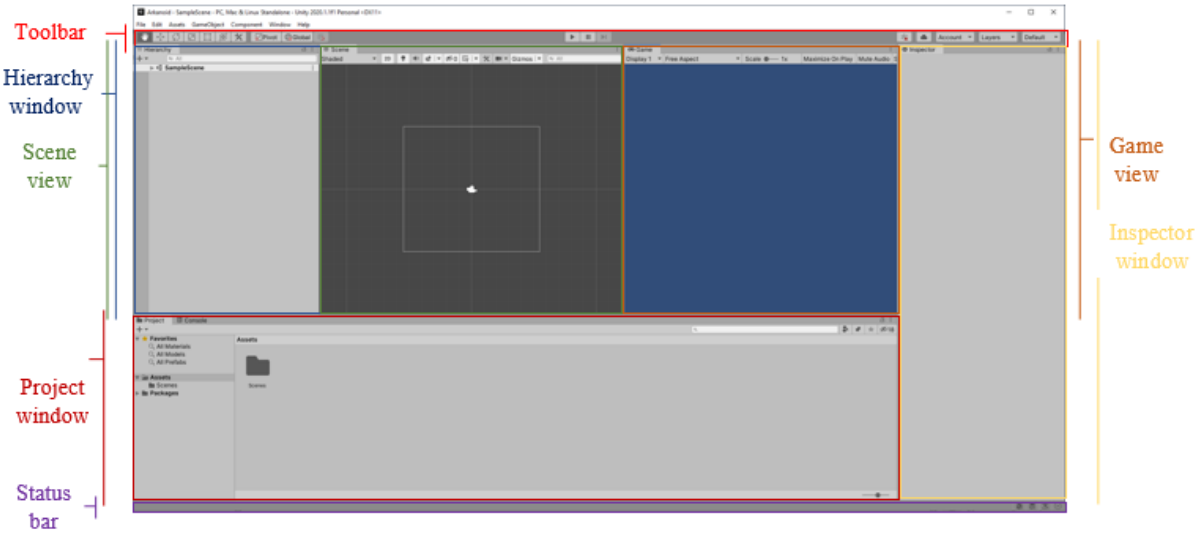
Obr. 3 Prechod jednotlivými scénami.

Nový projekt založíme pomocou Unity Hub kliknutím na tlačidlo NEW. Zvolíme 2D typ hry, určíme jej názov → *Arkanoid* a zvolíme umiestnenie. Následne potvrdíme kliknutím na tlačidlo CREATE (Obr. 4).



Obr. 4 Vytvorenie nového projektu.

Možnú výslednú podobu prázdneho projektu zobrazuje obrázok 5:



Obr. 5 Zobrazenie prázdneho projektu v Unity.

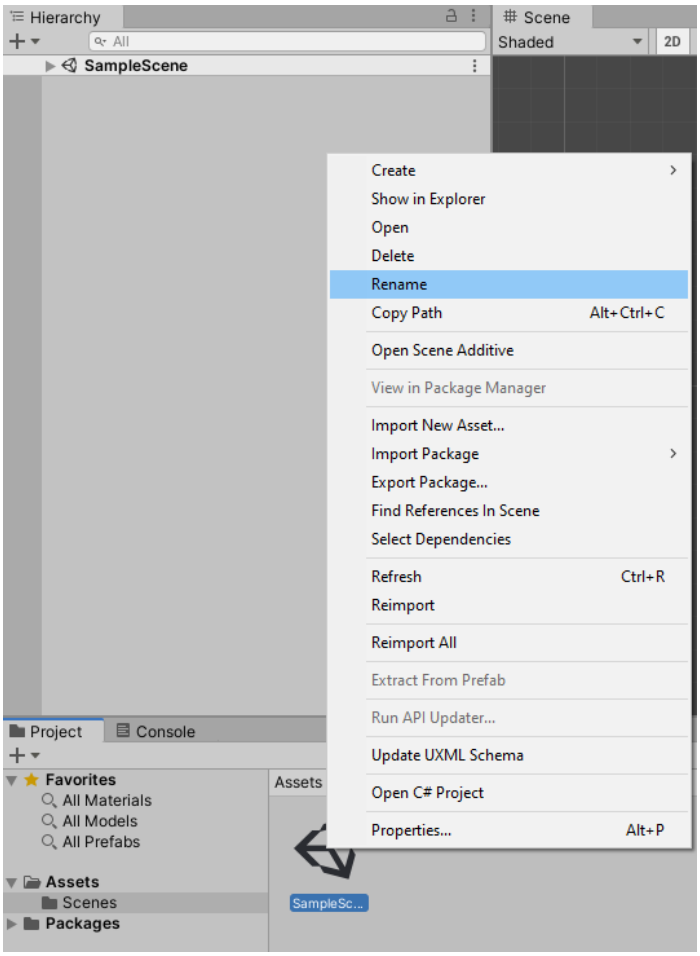
Rozloženie jednotlivých okien záleží od vašich preferencií, preto sa tento pohľad môže mierne líšiť v závislosti od nastavení rozloženia prostredia, t. j. *Layout-u*. Pri prvotnom vytváraní projektov často využívam také rozloženie, ako vidíte na obrázku (máme k dispozícii – sú viditeľné všetky základné okná):

- *Toolbar* – nachádza sa v hornej časti okna. Pomocou neho máme prístup k základným funkcionalitám, ako sú napr. *Move Tool* – pre zmenu pozície objektu; *Rotate Tool* – pre zmenu rotácie objektu; *Scale Tool* – pre zmenu veľkosti objektu resp. *Rect Tool* – pre zmenu pozície, veľkosti aj rotácie objektu. V jeho ľavej časti sa nachádzajú základné nástroje pre manipuláciu s objektmi v *Scene view*. V strede sa nachádzajú tlačidlá *Play*, *Pause* a *Step*. Tlačidlá napravo sprístupňujú prácu s *Unity Collaborate*, *Unity Cloud Services* a vašim *Unity kontom*, nasledované tlačidlami *Layers* a *Layout*, ktoré slúžia pre určenie viditeľnosti vrstiev, ako aj možností nastavenia rozloženia prostredia, alebo definovania vlastného.
- *Hierarchy window* – nachádza sa na ľavej strane okna. Tu sa nachádza hierarchicky usporiadaná textová reprezentácia každého objektu použitého v scéne.

- *Scene view* – nachádza sa napravo od okna *Hierarchy*. Tu môžeme vizuálne nastavovať scénu. Scénu si môžeme zobraziť v 2D, resp. 3D perspektíve, vzhľadom na typ projektu.
- *Game view* – sa nachádza hneď vedľa *Scene view*. Poskytuje náhľad ako bude vyzerať vaša hra po renderovaní vzhľadom na použitú kameru (*Scene Camera*). Po stlačení tlačidla *Play* sa spustí simulácia.
- *Inspector window* – nachádza sa na pravej strane. Pomocou tohto okna si môžeme prezerať, alebo nastavovať jednotlivé vlastnosti práve vybratého objektu. V závislosti od typu zvoleného objektu sa obsah tohto okna mení.
- *Project window* – nachádza sa v spodnej časti okna. Zobrazuje obsah (*assets*), ktorý je vo vašej hre prístupný. Kedykoľvek importujete *asset* do hry, zobrazí sa v tomto okne. Súčasťou okna je aj záložka *Console*, v ktorej môžeme sledovať chybové hlásenia.
- *Status bar* – nachádza sa v spodnej časti okna. Poskytuje notifikácie o rôznych Unity procesoch a rýchly prístup k relevantným nástrojom a nastaveniam (Unity Documentation, 2020a).

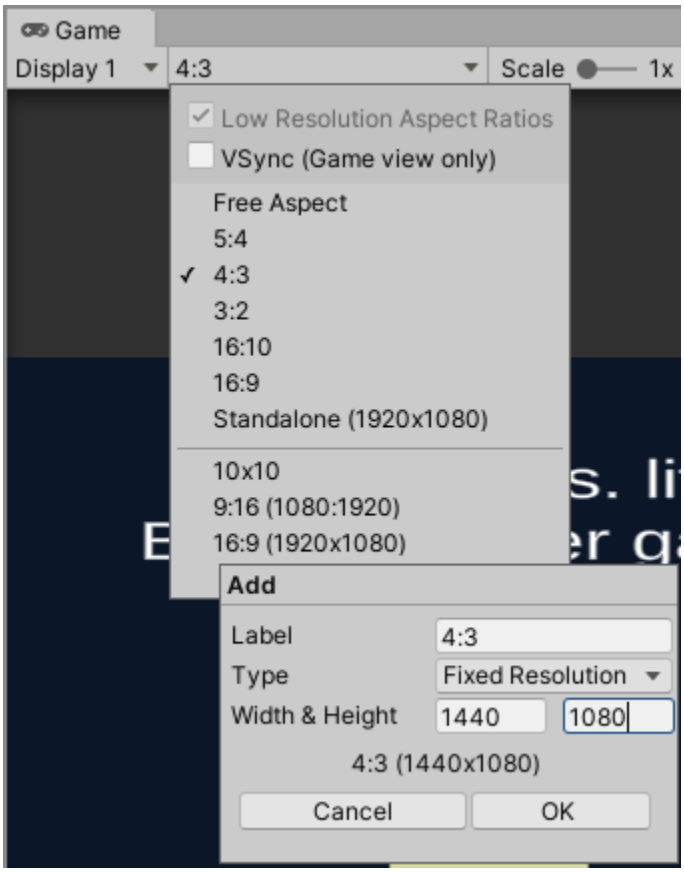
Ak potrebujete detailnejšie informácie týkajúce sa inštalácie Unity, začatia práce v ňom ako aj prestavenia prostredia odporúčam siahnuť priamo po dokumentácii Unity.

Aktuálnu scénu, ktorej názov je *SampleScene* premenujeme na *StartMenu*. Urobíme to tak, že v okne *Project*, v priečinku *Assets* a podpriečinku *Scenes* nájdeme túto scénu a pomocou kontextovej ponuky a možnosti *Rename* ju premenujeme (Obr. 6).



Obr. 6 Premenovanie existujúcej scény.

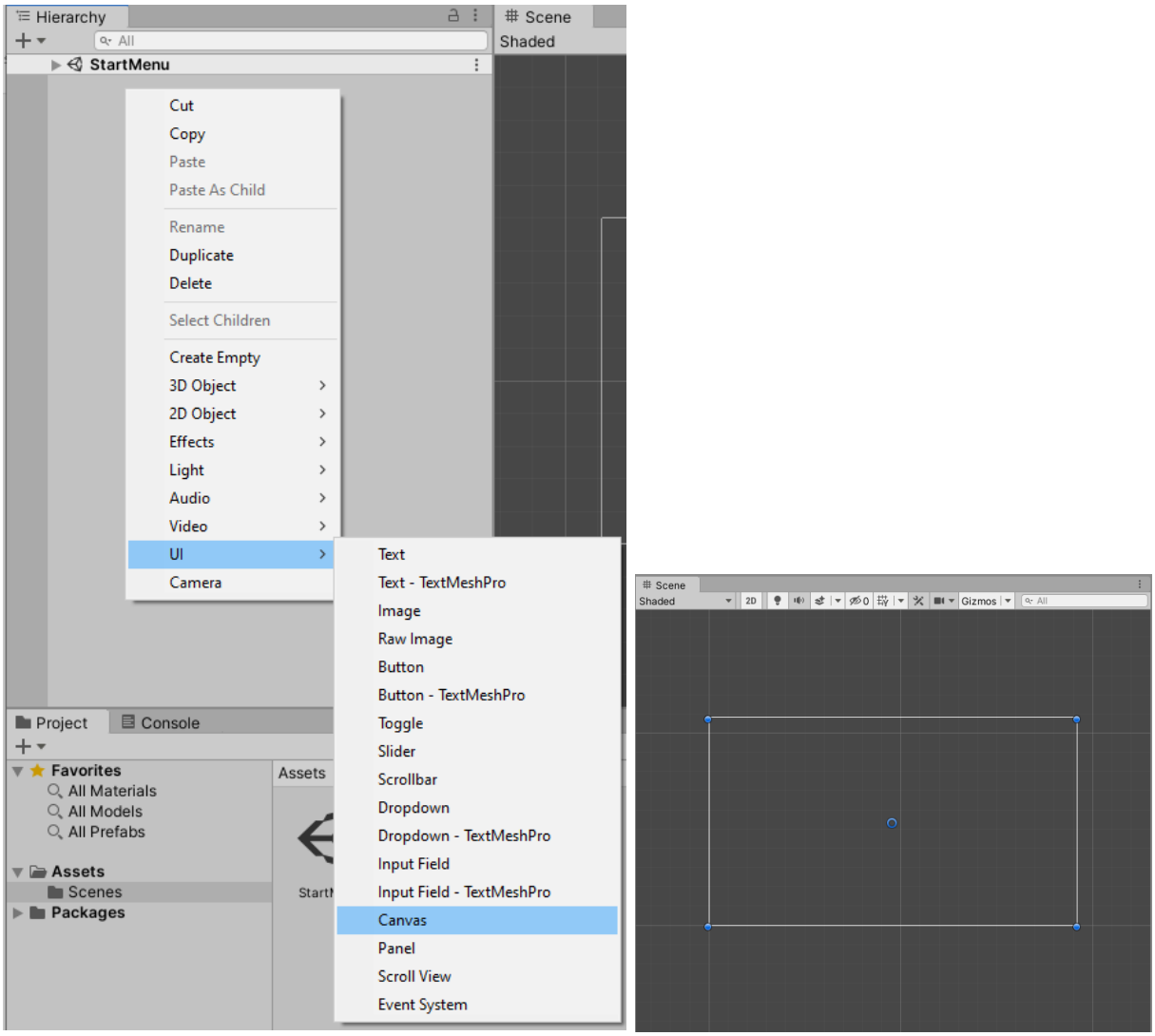
Pre túto hru odporúčame ponechať *Aspect Ratio* na 4:3 a rozlíšenie (*resolution*) nastaviť na 1440 x 1080 (Obr. 7). To ovplyvní rozlíšenie hry. Podľa Stewarta (2020) je nastavenie pre *Aspect Ratio* na 16:9 s rozlíšením 1920 x 1080 jedným z najčastejšie používaných, ale nie jediným.



Obr. 7 Nastavenie pre Aspect Ratio a rozlíšenie.

2.1 *UI element – Canvas*

Do scény, konkrétne do okna *Hierarchy* pridáme objekt *Canvas* z ponuky *UI* = *User Interface*, aby sme vytvorili priestor, kde budeme pridávať ďalšie objekty tvoriace používateľské rozhranie. Vo všeobecnosti platí, že *UI elementy* musia byť potomkami *Canvasu*. *Canvas area* je zobrazená ako obdĺžnik, ktorý vidíme v *Scene view*. Jedným zo základných nastavení *Canvasu* je *Render Mode*, ktorý ovplyvní, ako sa *Canvas* zobrazí na obrazovke (Obr. 8).

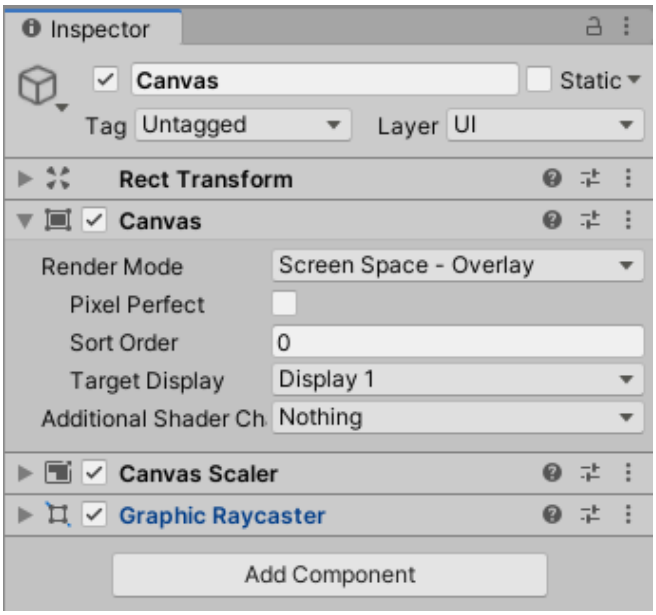


Obr. 8 Pridanie *Canvas* a jeho zobrazenie v *Scene view*.

Rozlišujeme tri módy:

1. *Screen Space – Overlay* – tento mód renderuje *UI elementy* na vrchu scény. V prípade ak dôjde k zmene veľkosti alebo k zmene rozlíšenia projektu, *Canvas* automaticky prispôsobí svoju veľkosť. Aj my v našom projekte ponechávame toto nastavenie.
2. *Screen Space – Camera* – ide o obdobné nastavenie ako *Screen Space – Overlay*, avšak zohľadňuje nastavenia kamery, na ktoré reflektujú použité *UI elementy*.
3. *World Space* – pri tomto móde sa *Canvas* správa ako každý iný objekt v scéne. Veľkosť canvasu je daná manuálne, je renderovaný pred, resp. za objektmi v scéne podľa jeho umiestnenia. Tento mód je vhodný v prípade ak

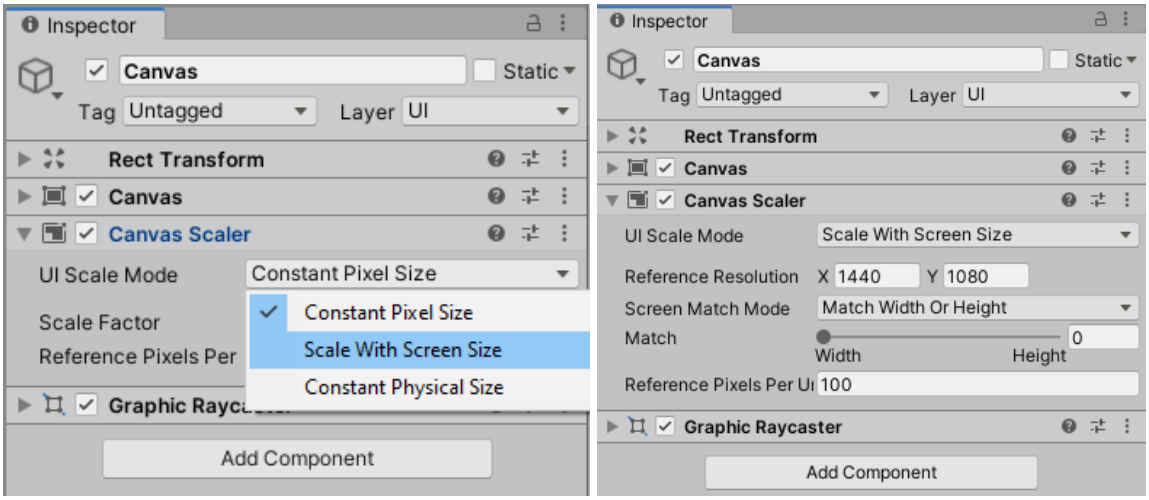
napr. ponuka hry (menu) má byť časťou herného sveta, t. j. toto menu má byť stále prístupné.



Obr. 9 Nastavenie Render Mode.

2.1.1 *UI Anchors*

Ak pracujeme s *Canvas* je dôležité mať nastavenú vlastnosť *UI Scale Mode* na *Scale With Screen Size*, pretože potom sa jednotlivé objekty prispôsobujú podľa zmeny veľkosti obrazovky. *Reference Resolution* nastavíme na 1400 x 1080, ako sme uviedli vyššie (Obr. 10).

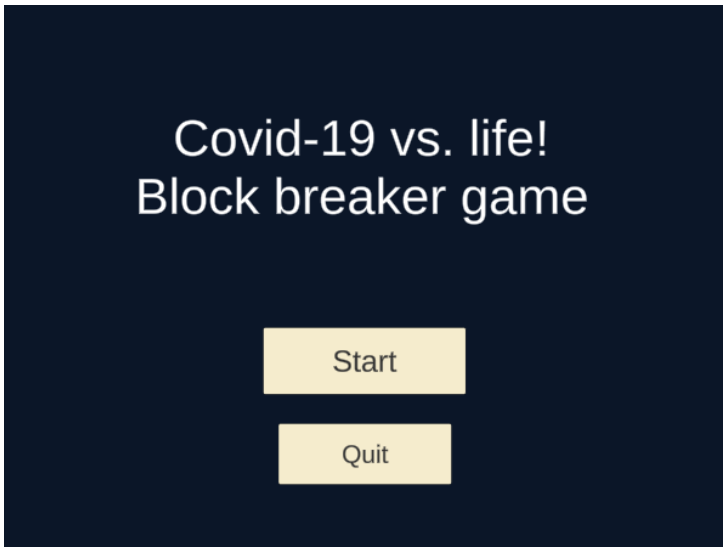


Obr. 10 Nastavenie *UI Scale Mode* a *Reference Resolution*.

Každému jednému objektu by sme mali nastaviť tzv. ukotvenie, t. j. relatívnu pozíciu v okne. Aj keď to možno nevyrieši všetky problémy súvisiace so želaným zobrazením jednotlivých elementov pri zmene veľkosti okna, je viac ako vhodné to používať.

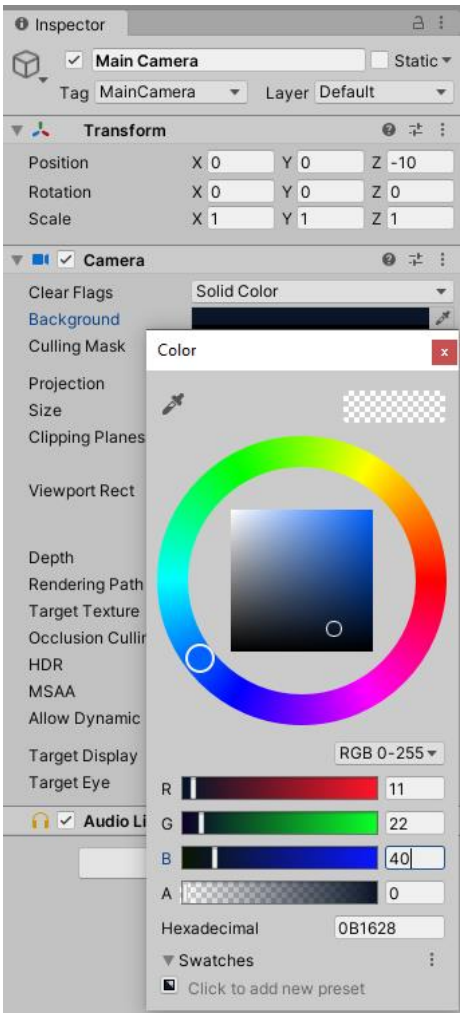
Zabezpečenie ukotvenia pozície komponentov pri zmene veľkosti okna spočíva v nasledujúcich krokoch. Podľa toho ktorú možnosť zvolíme, tak si to bude uchovávať relatívnu pozíciu voči príslušnej kotve. Napríklad, ak použijeme *left-bottom* bude sa to držať ľavého spodného rohu. Vysvetlíme si to na konkrétnych príkladoch objektov, ktoré budeme pridávať do *Canvasu*.

Cieľom je vytvoriť jednoduchú úvodnú obrazovku ako vidíme na obrázku 11.



Obr. 11 Úvodná obrazovka hry – scéna *StartMenu*.

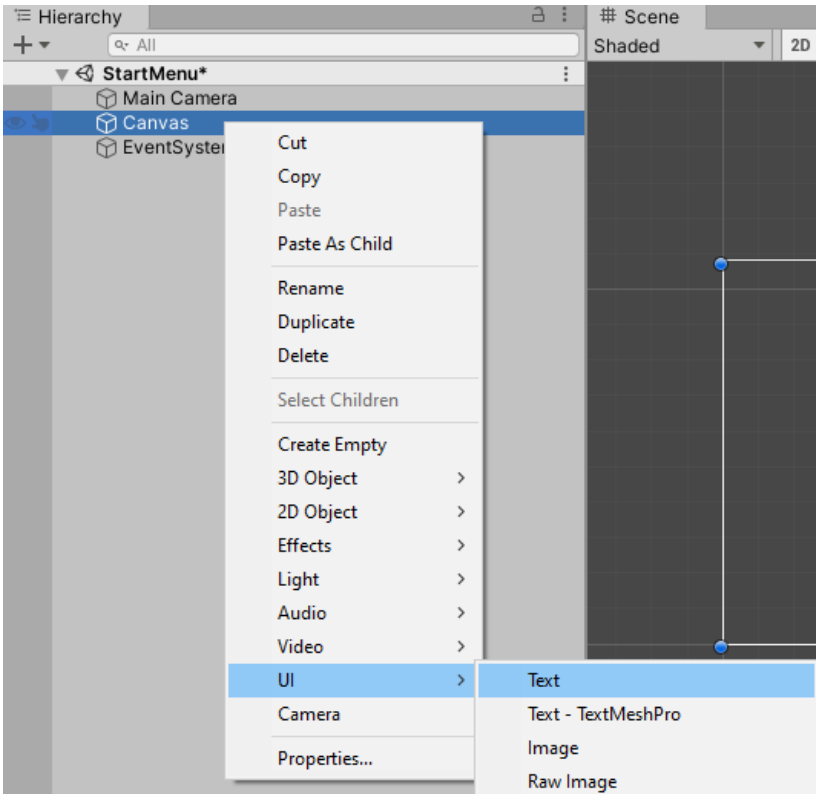
V tomto prípade nepoužívame žiadne pozadie. Zmenu farby pozadia sme dosiahli zmenou vlastnosti objektu kamera. Ide o vlastnosť *Background*, kde sme farbu vďaka palete nastavili na nami požadovanú (Obr. 12):



Obr. 12 Zmena nastavenia vlastnosti *Background* pre kameru.

2.1.2 *Text* a *TextMeshPro*

Prvý prvok, ktorý pridáme na úvodnú obrazovku je Text. Tento môže byť použitý pre zobrazenie nadpisu, opisu, inštrukcií pre hráča a pod. Unity ponúka dve možnosti pridania textu: *Text* (Obr. 13) a *TextMeshPro* (TMP). Principiálny rozdiel medzi nimi je v tom, že TMP je dokonalejšia vizuálna podoba textu, ktorá poskytuje viacero možností na jeho úpravu a nastavenia vzhľadom na jeho vizuál, štýl a textúru (Unity, 2020b; Unity, 2020c).

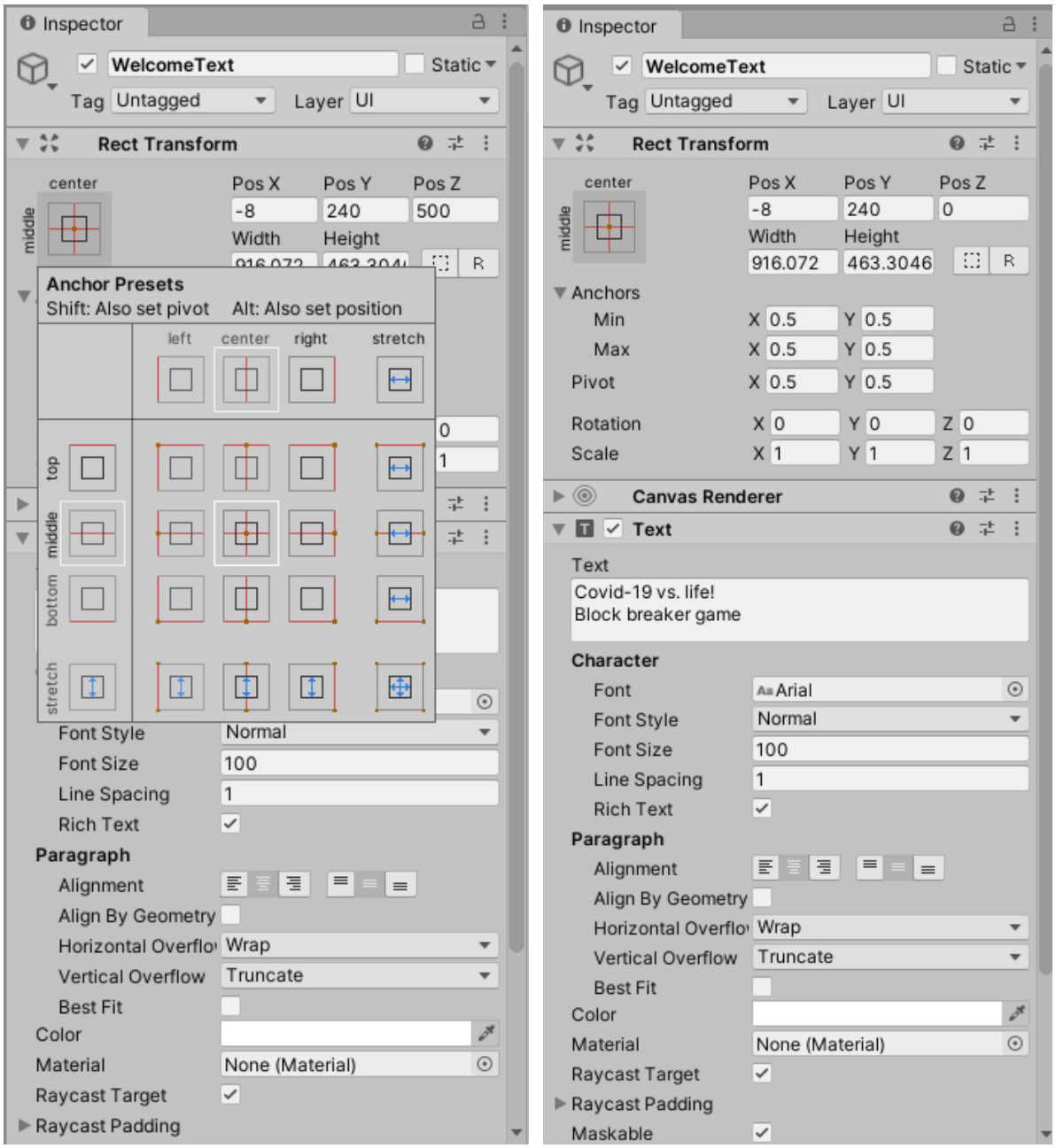


Obr. 13 Pridanie UI *elementu – Text*.

Danému objektu ako prvé nastavíme pomenovanie *WelcomeText*, čo robíme v okne *Inspector*. Následne pomocou ponuky *Rect Transform* tiež v okne *Inspector*, ukotvíme tento objekt do stredu obrazovky. Cieľom je, aby aj pri zmene veľkosti okna projektu, tento ostal umiestnený v strede. Následne nastavíme základné vlastnosti pre text ako sú: *Text* – kde píšeme text, ktorý sa nám má na obrazovke zobraziť, *Font*, *Font Style*, *Font Size* – túto zväčšujeme, aby bol text dostatočne viditeľný. Pri zmene veľkosti bude pravdepodobne potrebné, zmeniť aj veľkosť daného objektu pomocou nástroja *Scale Tool*, ktorý nájdeme v *Toolbare*. Ak by ste neupravili veľkosť objektu, môže sa stať, že po zmene veľkosti text neuvidíte na obrazovke, pretože je príliš veľký pre danú veľkosť objektu.

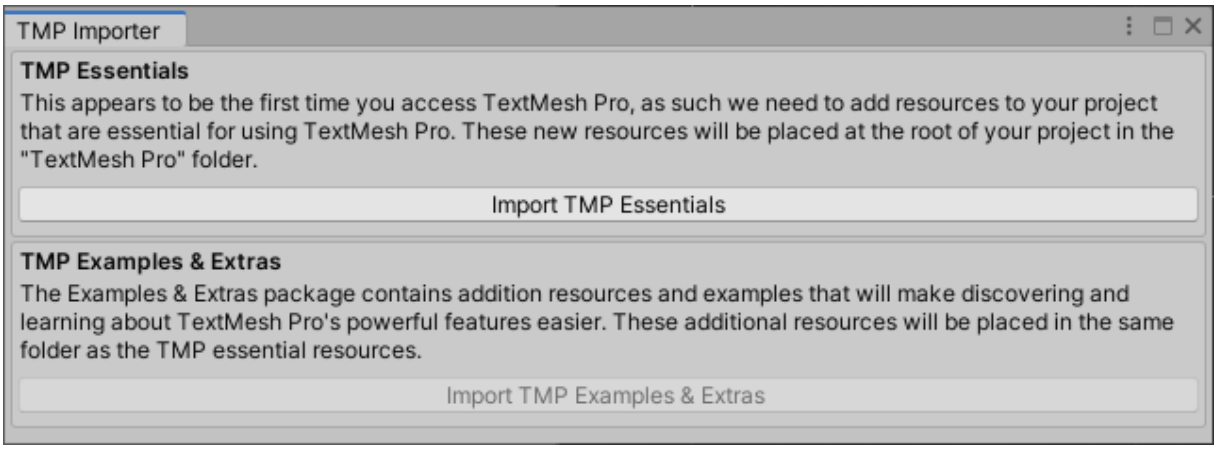
Ďalšiu vlastnosť, ktorú upravujeme je *Alignment* – využívame možnosť *Center* a tiež zmeníme farbu (*Color*) na bielu. Ako posledné nastavíme pozíciu daného objektu, t. j. jeho x a y súradnicu tak, aby sa nachádzal približne v strede obrazovky. Nastavenie v smere osi z zatiaľ nie je pre nás smerodajné. Javí sa to samozrejmé, keďže sa venujeme vývoju hry v 2D priestore, ale aj v 2D priestore budeme

pracovať s touto osou (viac v kapitole 3). Jednotlivé nastavenia sú samozrejme v kompetencii vývojára. Použité nastavenia je možné vidieť na obrázku 14.



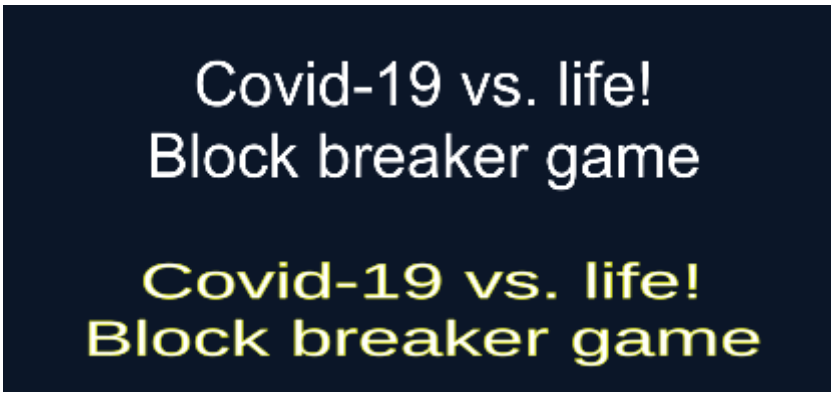
Obr. 14 Nastavenia vlastností objektu *Text*.

Ak by sme s vizuálnou reprezentáciou textu neboli spokojný, môžeme siahnuť po pridaní objektu typu *TextMeshPro*, ktorý poskytne širšie možnosti vizuálnej reprezentácie textu. Pri prvotnom pridaní bude nutné importovať zodpovedajúce súčasti, na čo vás Unity vyzve (Obr. 15).

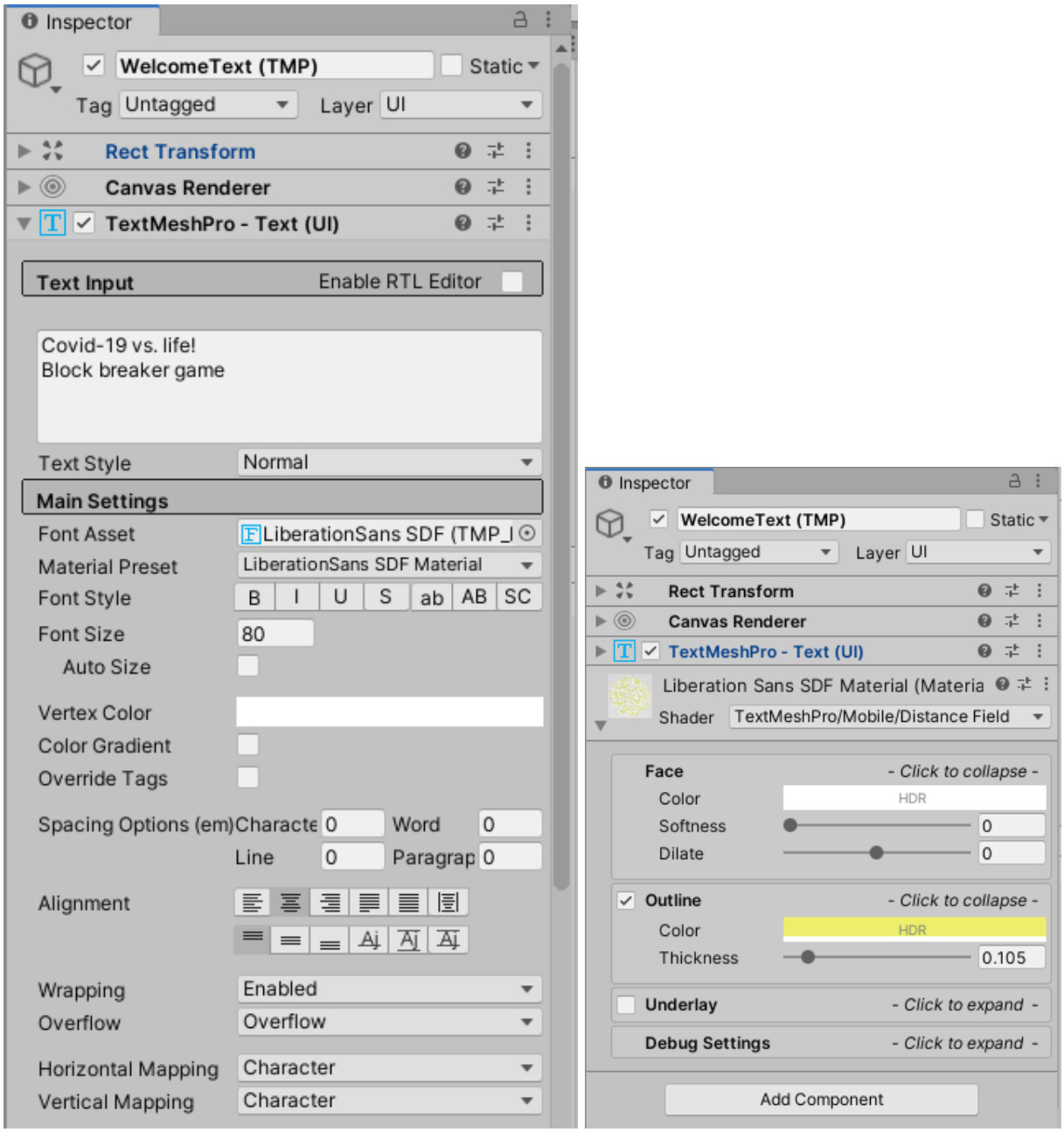


Obr. 15 Importovanie súčastí pre použitie objektu *TextMeshPro*.

Základné vlastnosti pre tento objekt sú obdobné, ako pre objekt *Text*. Ponúka však rôzne rozšírenia. Napríklad my sme využili možnosť ohraničiť použitý text (vlastnosť *Outline*) a zvolila som pre to žltú farbu. Tiež som zmenila *font* (Obr. 16). Je zrejmé, že vizuálna reprezentácia nadpisu je efektnejšia.



Obr. 16 Vizuálna reprezentácia nadpisu pomocou *Text* a *TextMeshPro*.

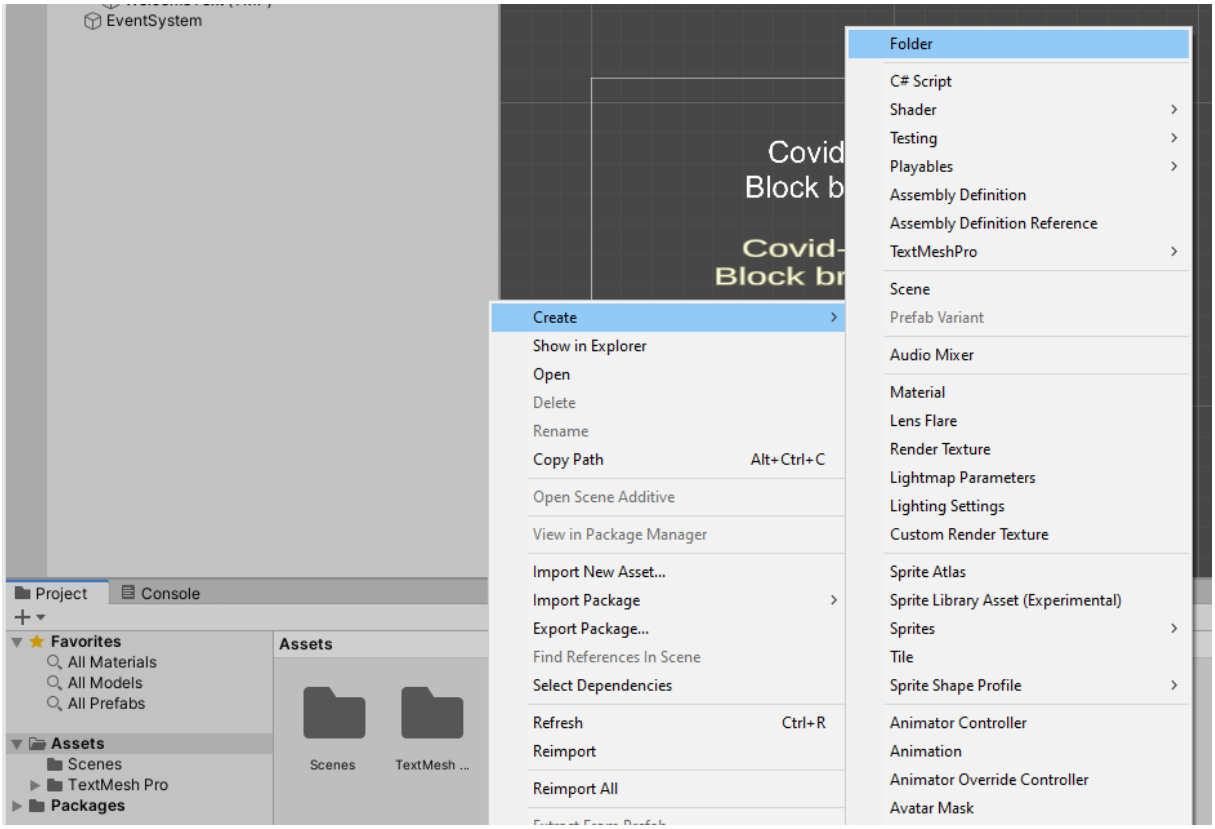


Obr. 17 Nastavenia vlastností *TextMeshPro*.

Ďalšou zmenou, ktorú vieme pri Unity v súvislosti s textom využiť je zmena fontu, pričom nemusíme využívať len preddefinované fonty, ale môžeme použiť vlastný font. Ako zdroj pre rôzne fonty je možné využiť webové sídlo: https://www.dafont.com/

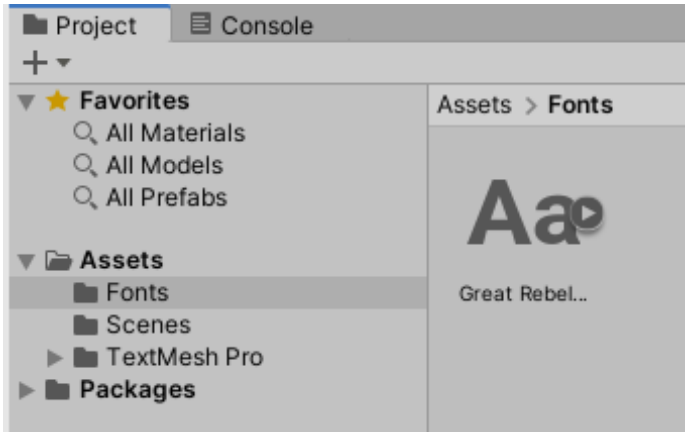
Pre účely tejto ilustrácie sme využili font s názvom *Great Rebellion*, ktorý má mierne hororový nádych a vzhľadom na tematiku hry sa javí ako vhodný. Skôr ako si daný font pridáme do projektu, je dobré od začiatku udržiavať v projekte

poriadok. Z tohto dôvodu v okne *Project* pomocou kontextovej ponuky vytvoríme nový priečinok (*Folder*), ktorý pomenujeme *Fonts* (Obr. 18).



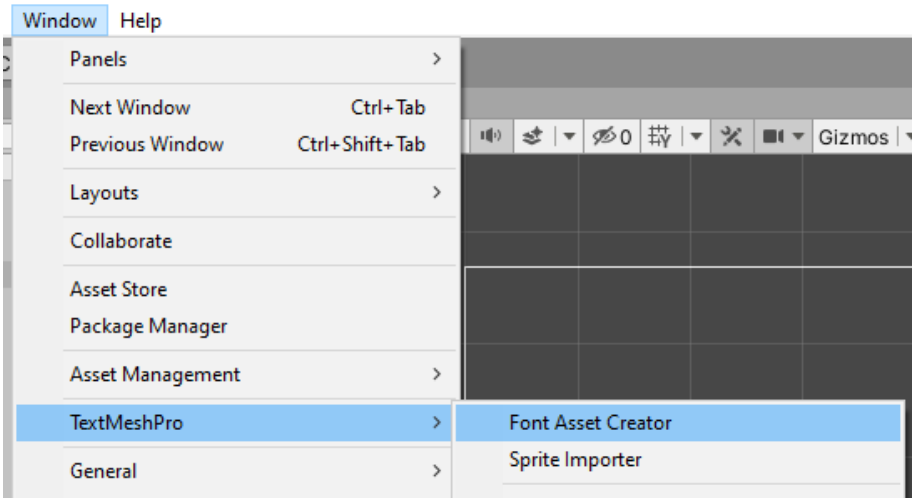
Obr. 18 Vytvorenie prázdneho priečinku v okne *Project*.

Do tohto priečinku umiestnime daný font, jeho jednoduchým presunutím zo zdrojového priečinku. Výsledok je zobrazený na obrázku 19:



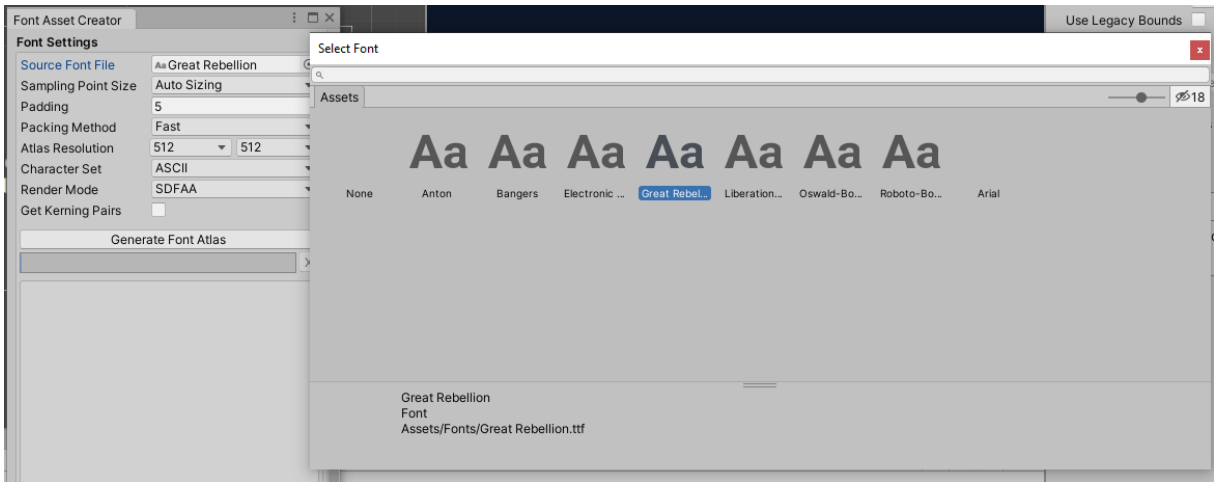
Obr. 19 Pridanie vlastného fontu do projektu.

Aby sme mohli daný font použiť, musíme ho ešte vytvoriť pomocou ponuky *Window* → *TextMeshPro* → *Font Asset Creator* (Obr. 20).



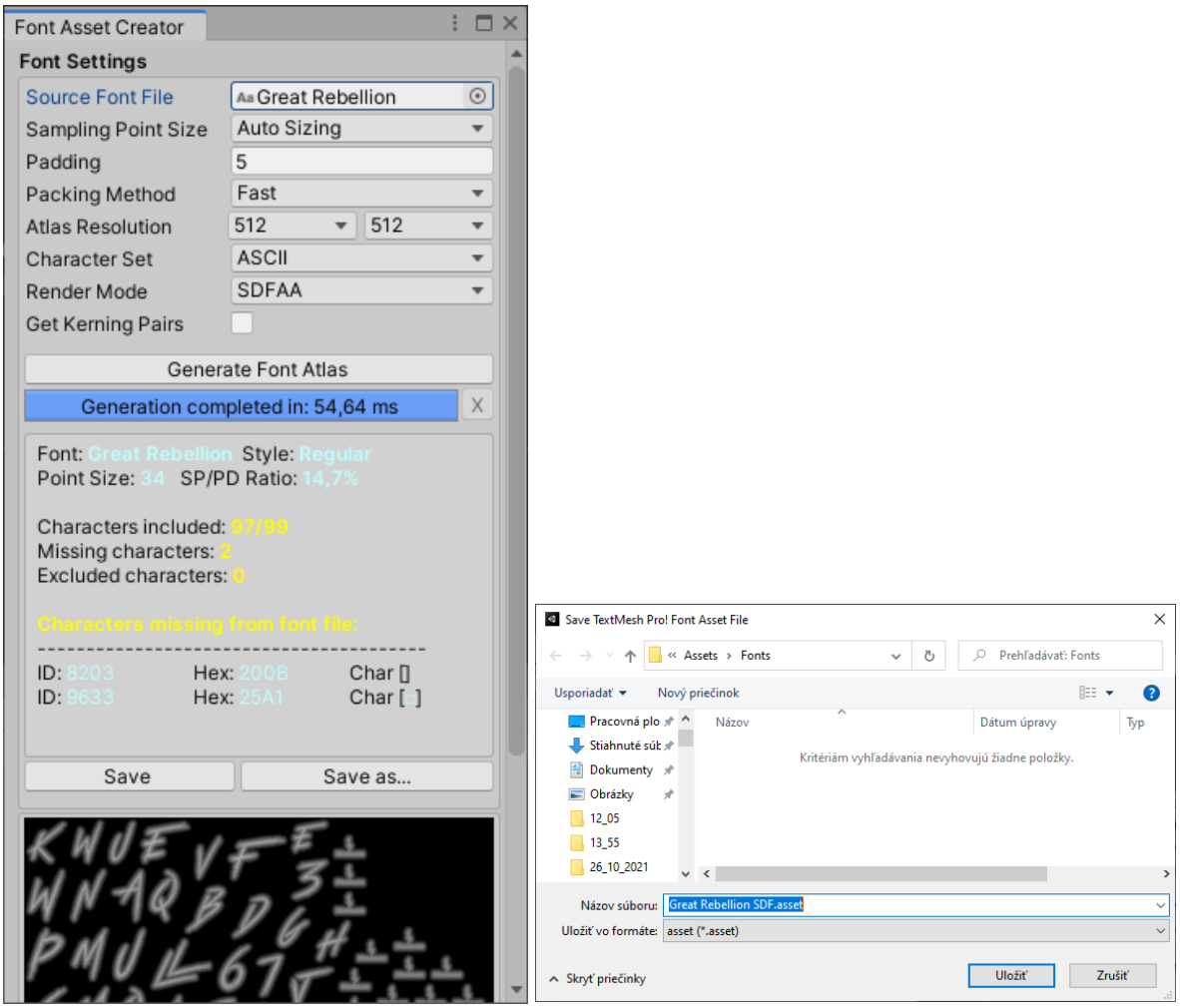
Obr. 20 Vytvorenie nového fontu.

Ako prvý krok v možnosti *Source Font File* zvolíme font, ktorý chceme použiť:



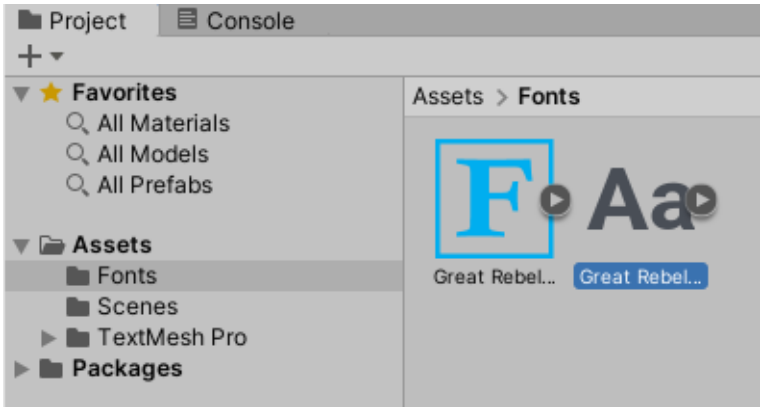
Obr. 21 Voľba fontu.

Následne ho necháme vygenerovať pomocou možnosti *Generate Font Atlas* a uložíme do pripraveného priečinku s názvom *Fonts* (Obr. 22).



Obr. 22 Vygenerovanie a uloženie fontu.

Výsledkom je nový font pridaný do priečinku *Fonts* v našom projekte, ktorý už môžeme používať:



Obr. 23 Výsledok po pridaní nového fontu.

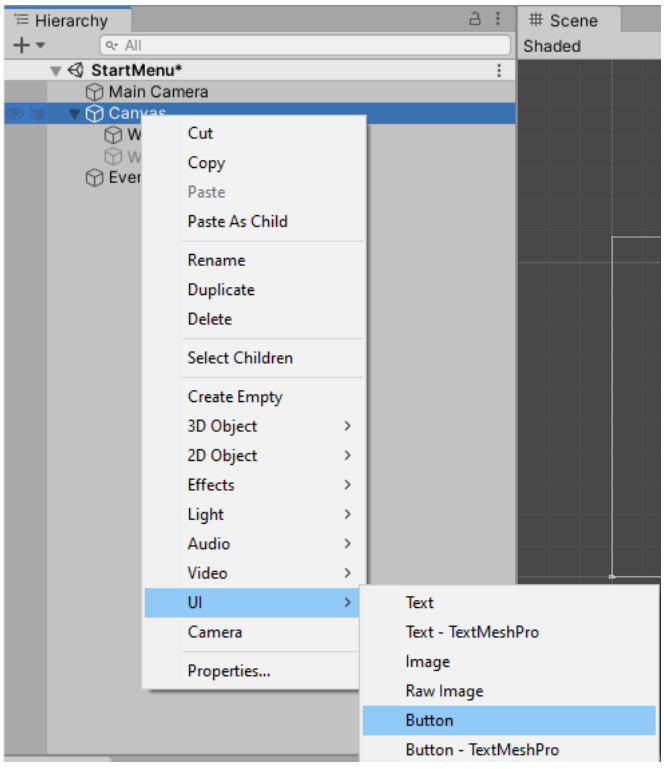
Zmena fontu pre náš *WelcomeText* na novo pridaný font sa prejaví vizuálne nasledovne:

Obr. 24 Nadpis s použitím nového fontu.

Tip autora: Pre tento font sme museli z nadpisu odstrániť interpunkčné znaky ako „.“, „–„ a „!“ z dôvodu, že tento font ich nepodporuje. Zobrazenie číslic tiež nie je najvhodnejšie, aj keď je efektné. Je preto vhodné sa s daným fontom detailne oboznámiť ešte pred jeho použitím. Nakoniec sme použili vstavaný font *LiberationSans SDF*.

2.1.3 Tlačidlo = *Button*

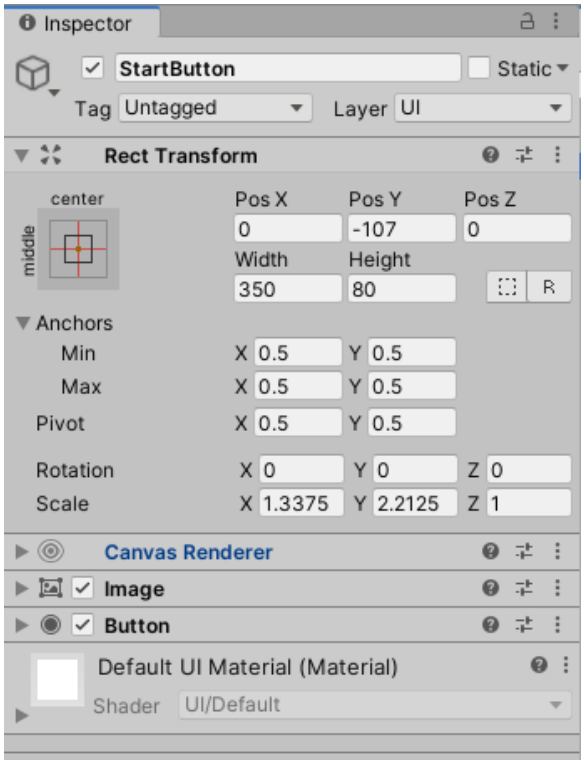
Ako ďalší prvok, pridáme objekt tlačidlo (Button), ktoré pomenujeme *StartButton*:



Obr. 25 Pridanie UI *elementu – Button*.

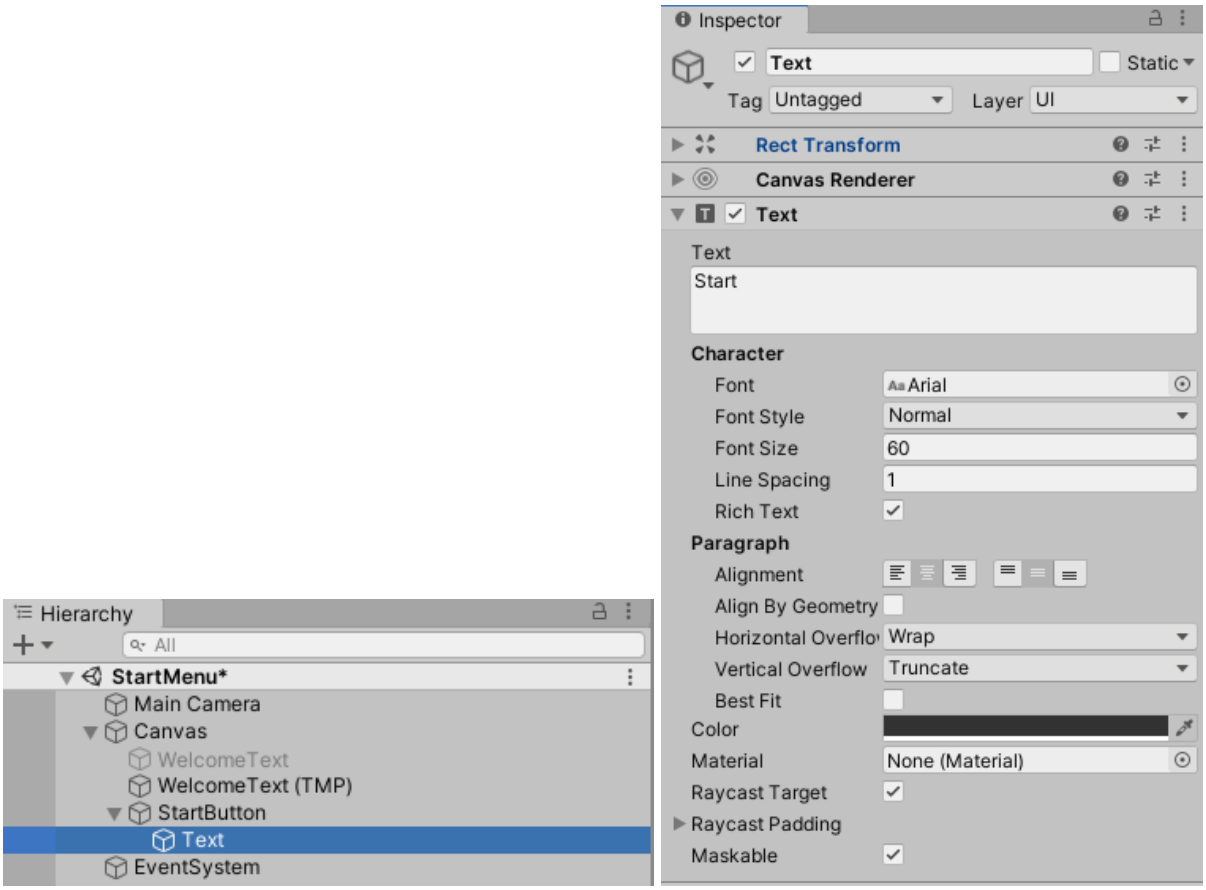
Pomocou nástrojov pre zmenu pozície a veľkosti nachádzajúce sa v *Toolbare* nastavíme požadovanú veľkosť tlačidla, ako aj jeho pozíciu (Obr. 26). Samozrejme,

tieto vlastnosti možno ovplyvňovať aj priamo v *Inspectore*, pomocou nastavenia pozície v smere osi x = PosX, resp. y = PosY, zmenou šírky = *Width*, resp. výšky = *Height*:

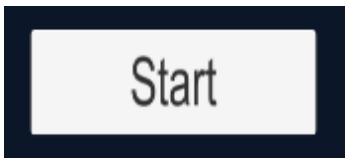


Obr. 26 Zmena veľkosti a pozície pre objekt typu tlačidla.

Rozbalením ponuky pre tlačidlo v okne *Hierarchy*, si môžeme všimnúť, že tento objekt má potomka, pomocou ktorého vieme nastavovať znenie textu ako aj vlastnosti textu, ktorý má byť na tlačidle zobrazený. V našom prípade tam chceme mať text *Start*, preto stačí vo vlastnosti *Text* v okne *Inspector* pre tento objekt zadať tento text, alebo upraviť jeho iné vlastnosti:

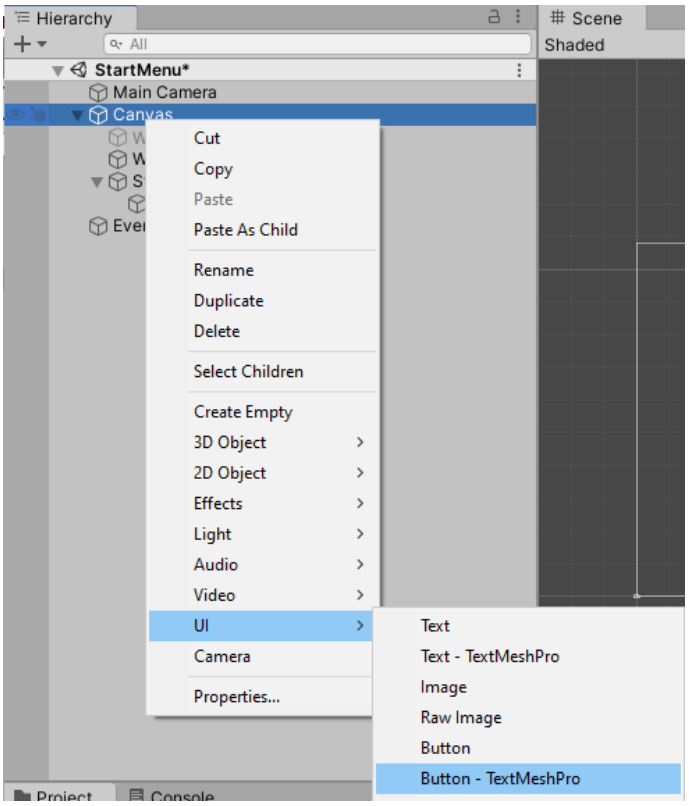


Obr. 27 Nastavenie textu pre tlačidlo.



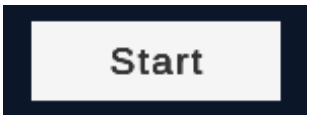
Obr. 28 Vizuálna reprezentácia tlačidla – *StartButton*.

V prípade ak chceme zachovať jednotnosť v používaných fontoch, či mať k dispozícii širšie možnosti nastavení, je možné pridať objekt *Button – TextMeshPro* (Obr. 28).



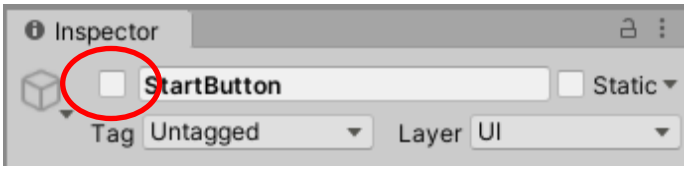
Obr. 29 Pridanie UI *elementu – Button – TextMeshPro*.

Obdobným spôsobom ako bolo vysvetlené vyššie nastavíme jednotlivé parametre a vlastnosti objektu. Výsledná vizuálna reprezentácia tlačidla by mohla vyzerať takto:



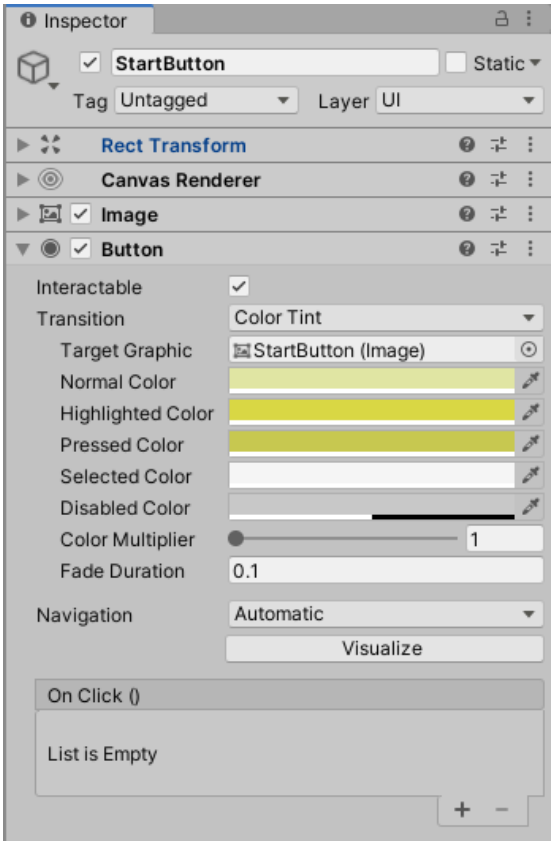
Obr. 30 Vizuálna reprezentácia tlačidla – *StartButton – TextMeshPro*.

Tip autora: V prípade ak sme pridali nejaký objekt, ktorý ale nechceme používať, ale tiež ho ešte nechceme definitívne vymazať, môžeme využiť jednoduchú možnosť jeho skrytia v scéne a to odškrtnutím zaškrtávacieho tlačidla (*checkboxu*) hneď veľa názvu daného objektu:



Obr. 31 Ako zneviditeľniť objekt v scéne

Nastavením farieb vo vlastnostiach objektu *Button* v okne *Inspector* vieme nastaviť správanie sa tlačidla pri normálnom stave = *Normal Color*, pri nájdení myškou nad neho = *Highlighted Color*, ako aj pri stlačení tlačidla = *Pressed Color*:

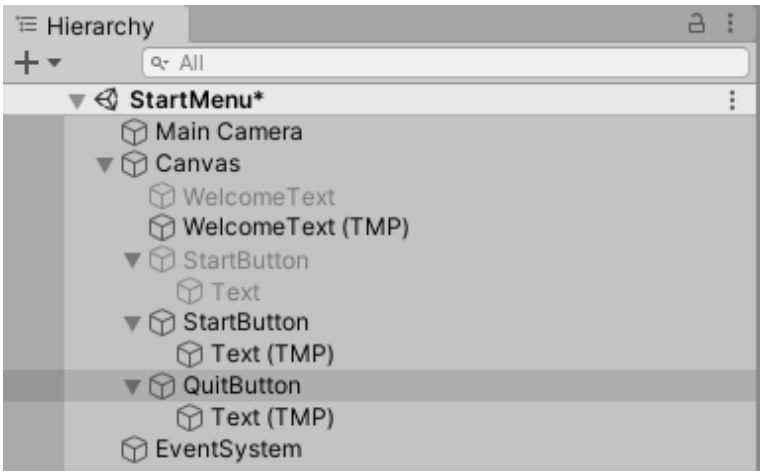


Obr. 32 Nastavenie farieb pre zvýraznenie jednotlivých stavov tlačidla.

V úvodnej scéne potrebujeme ešte jedno tlačidlo – *QuitButton*. Najjednoduchšie ho vytvoríme duplikovaním už existujúceho tlačidla a jeho úpravou. Stačí v okne *Hierarchy* vybrať existujúce tlačidlo a využiť kombináciu kláves Ctrl + D. Okrem zmeny textu sme zmenili aj veľkosť, z dôvodu dosiahnutia vyššej koncentrácie používateľa na tlačidlo *StartButton*.

Obr. 33 Výsledná vizuálna reprezentácia úvodného okna.

Štruktúra prvej scény vyzerá zatiaľ takto (s tým, že niektoré objekty, ktoré tvoria iné varianty objektov sme len skryli a zatiaľ nevymazali):



Obr. 34 Štruktúra prvej scény.

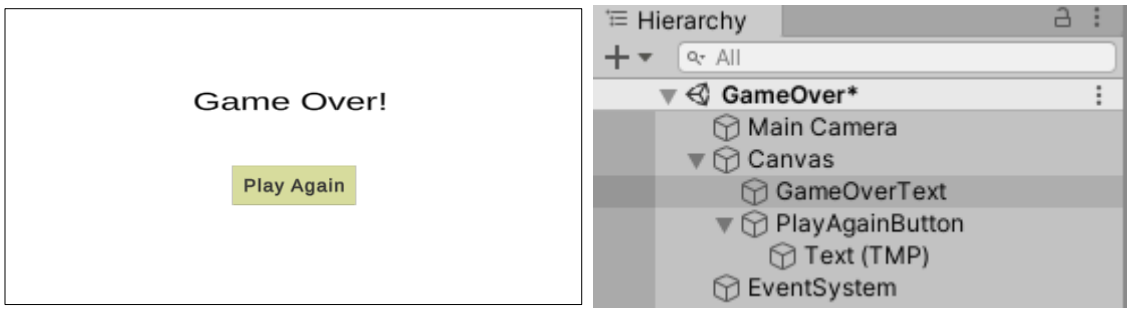
2.2 Vytvorenie scény

Opätovne, najjednoduchší spôsob vytvorenia druhej scény je vytvoriť ju duplikovaním už existujúcej. V *Assets* si nájdeme priečinok *Scenes*, vyberieme scénu *StartMenu* a stlačíme Ctrl + D. Novú scénu premenujeme na *Level_1*. Bude tvoriť jeden z levelov hry.

V tejto fáze postačí ak na scéne ponecháme len jedno z tlačidiel, ktoré pomenujeme *NextButton* a text tlačidla zmeníme na „Next“ (Obr. 35).

Obr. 35 Vizuál a štruktúra druhej scény.

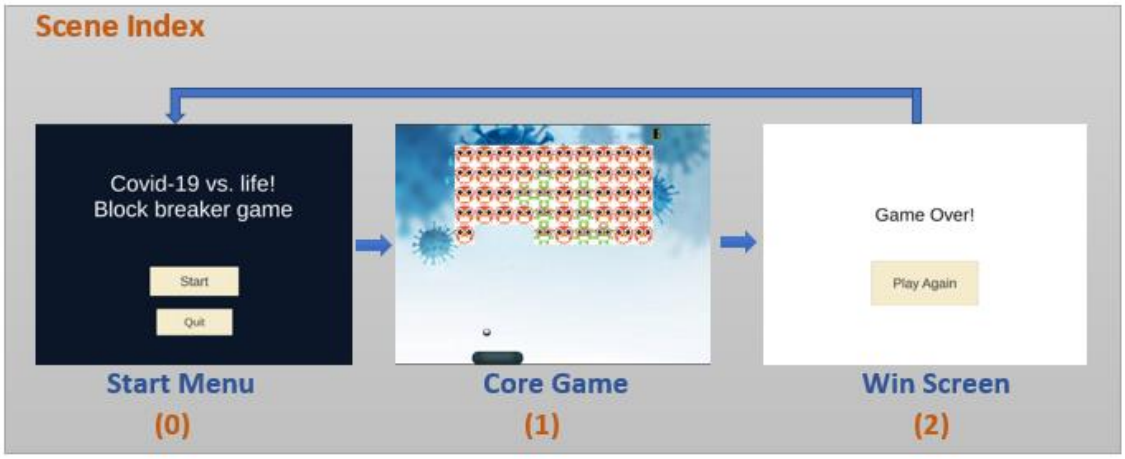
Poslednú, tretiu scénu vytvoríme obdobným spôsobom. Pomenujeme ju *GameOver*. Vizuál scény je zobrazený na obrázku 36. Na scéne sme ponechali text „Game Over!“ a tlačidlo *PlayAgainButton* s textom „Play Again“.



Obr. 36 Vizuál a štruktúra tretej scény.

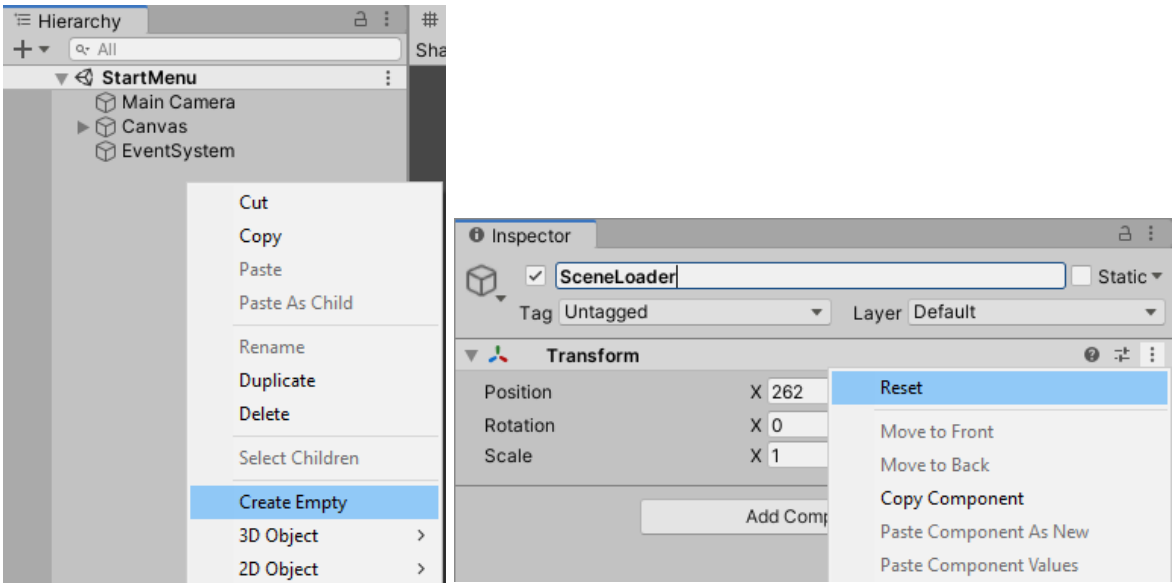
2.2.1 Načítanie scén pomocou tlačidla

Scény majú svoje indexy, pričom prvý index má hodnotu 0. Problémom pri používaní indexov scén môže byť, že v prípade ak zasiahneme do štruktúry hry dôjde k posunutiu indexov. Je preto vhodnejšie pri lineárnych prechodoch scénami, ak začneme na prvej snímke a postupujeme ďalej v budovaní scén tak ako majú scény nasledovať. Obrázok 37 zachytáva scény hry, ako aj prechod medzi nimi. Tento je lineárny, pričom z poslednej scény v prípade voľby používateľa sa môžeme vrátiť opäť na úvodnú scénu.

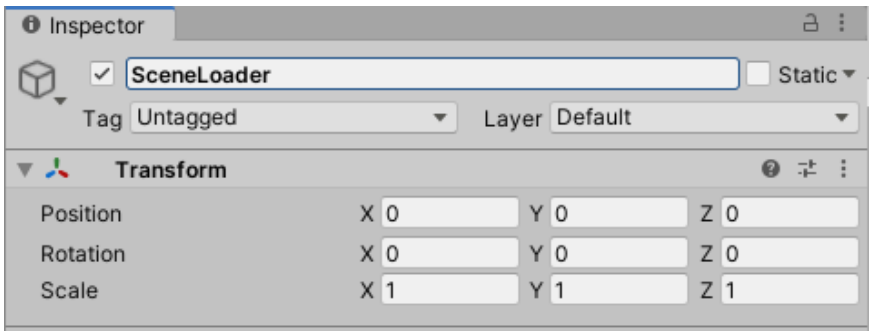


Obr. 37 Prechod scénami a ich indexy.

Pre zabezpečenie prechodu medzi snímkami pridáme v scéne *StartMenu* prázdny objekt (*Game object*), pomenujeme ho *SceneLoader* a resetneme nastavenia v okne *Inspector* v komponente *Transform* (je to dobrý zvyk pri prázdnych objektoch) (Obr. 38).

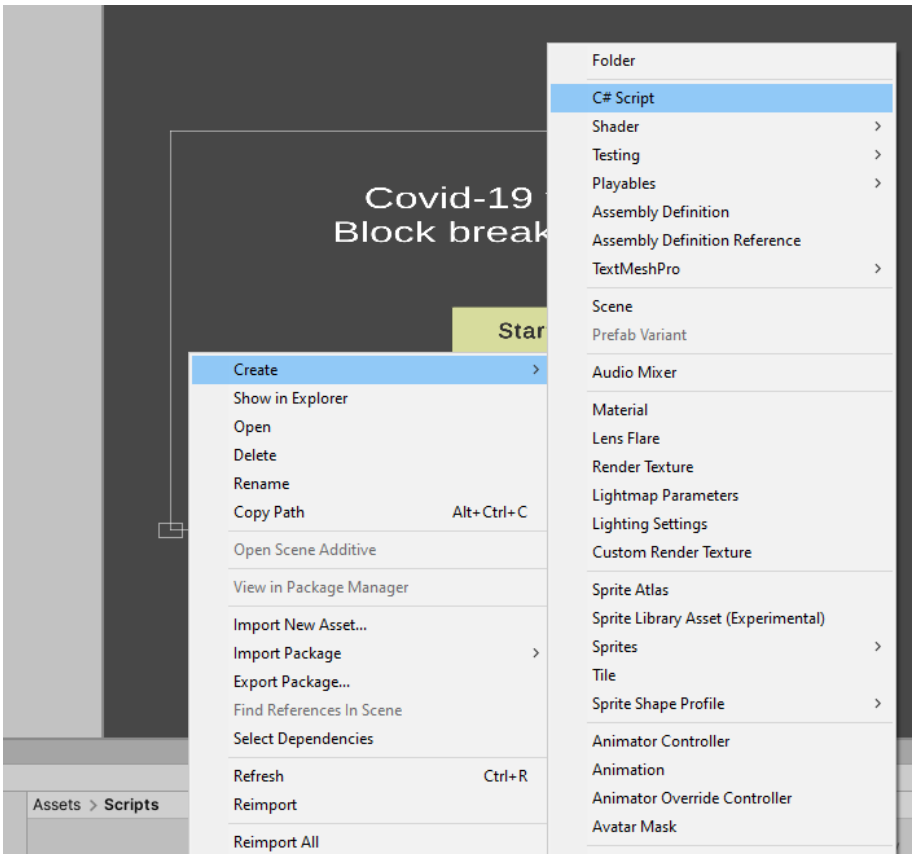


Obr. 38 Pridanie prázdneho objektu a resetnutie jeho vlastností.



Obr. 39 Výsledok pridania prázdneho objektu po resetovaní jeho vlastností.

Následne vytvoríme skript, ktorý sa bude starať o načítavanie scén. V *Assets* vytvoríme nový priečinok s názvom *Scripts* a využijeme kontextovú ponuku pre vytvorenie C# skriptu. Pomenujeme ho *SceneLoader*.



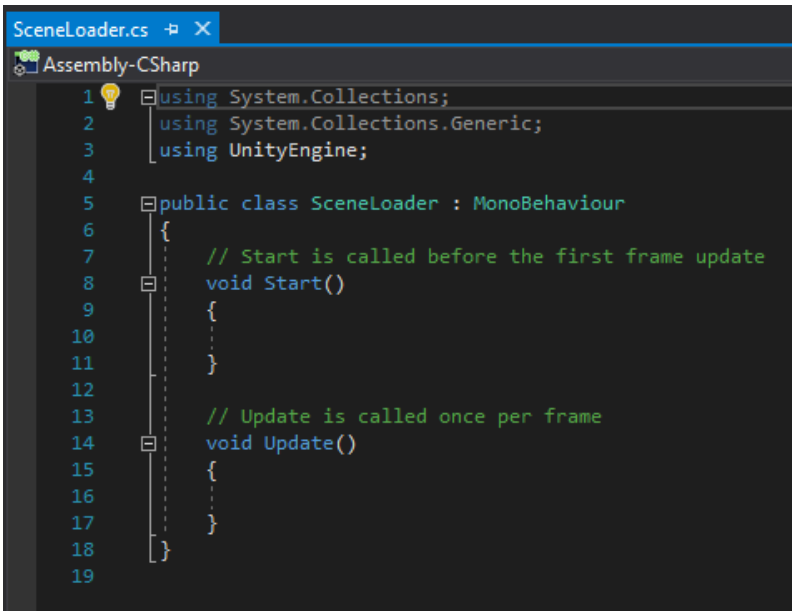
Obr. 40 Vytvorenie C# skriptu.

Skript otvoríme v aplikácii Visual Studio. Aby sme mohli pracovať so scénami a využívať preddefinované metódy, potrebujeme pridať *namespace*:

```
using UnityEngine.SceneManagement;
```

Pozn. autora: *Namespace* je ako adresár, ktorý združuje premenné, metódy, triedy a pod. Tá istá metóda sa môže nachádzať v rôznych *namespace*, preto je vhodné pridávať len tie *namespace*, ktoré potrebujeme pre danú prácu. *Namespace* nie je to isté čo *library*.

Obrázok 41 zobrazuje štruktúru prázdneho C# skriptu otvoreného vo Visual Studio.



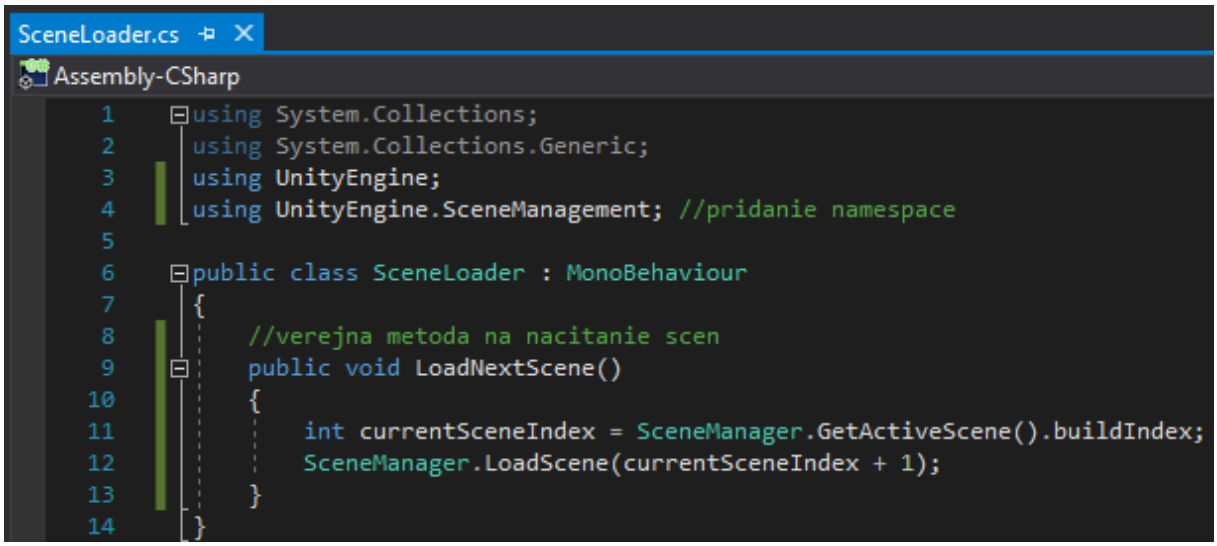
Obr. 41 Štruktúra prázdneho C# skriptu.

Funkciu *Start*() ani *Update*() v tomto prípade nebudeme potrebovať – môžeme ich teda vymazať. Vo všeobecnosti platí, že funkcia *Start*() sa vyvolá hneď na začiatku a len raz, takže slúži na nejaké východiskové nastavenia. Funkcia *Update*() je volaná každý *frame*. Bežne sa zvykne prehrávať 12 alebo 24 snímok za sekundu (*frame per second*). Nastavenia sú však individuálne, s čím sa môžeme stretnúť najmä pri tvorbe animácií.

Vytvoríme si verejnú metódu, ktorá sa bude starať o načítavanie scén. Pomenujeme ju *LoadNextScene*(). Pre zistenie hodnoty indexu aktuálnej scény využijeme metódu *GetActiveScene()*, ktorá vráti aktuálnu scénu a vlastnosť (*property*) *buildIndex* pomocou ktorej získame priamo hodnotu indexu tejto scény. Túto hodnotu si uložíme do lokálnej celočíselnej premennej s názvom *currentSceneIndex*. Pre načítanie scény využijeme metódu *LoadScene()*, v ktorej môžeme použiť priamo index, alebo názov danej scény. Zmeny v skripte uložíme.

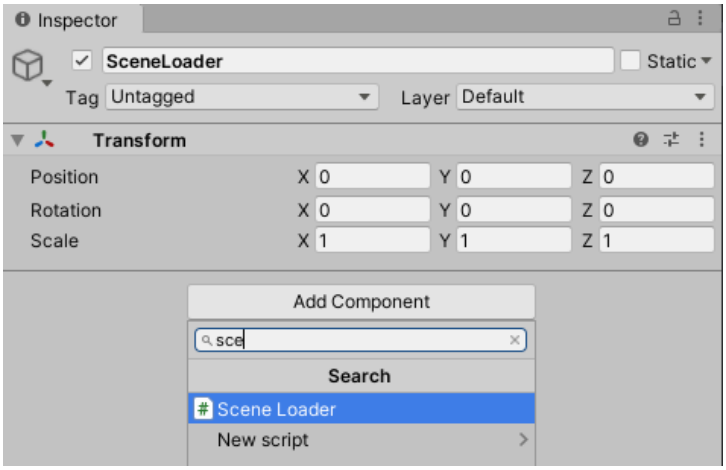
```
public void LoadNextScene()
{
    int currentSceneIndex = SceneManager.GetActiveScene().buildIndex;
    SceneManager.LoadScene(currentSceneIndex + 1);
}
```

Obrázok 42 zachytáva celkový pohľad na tento skript.

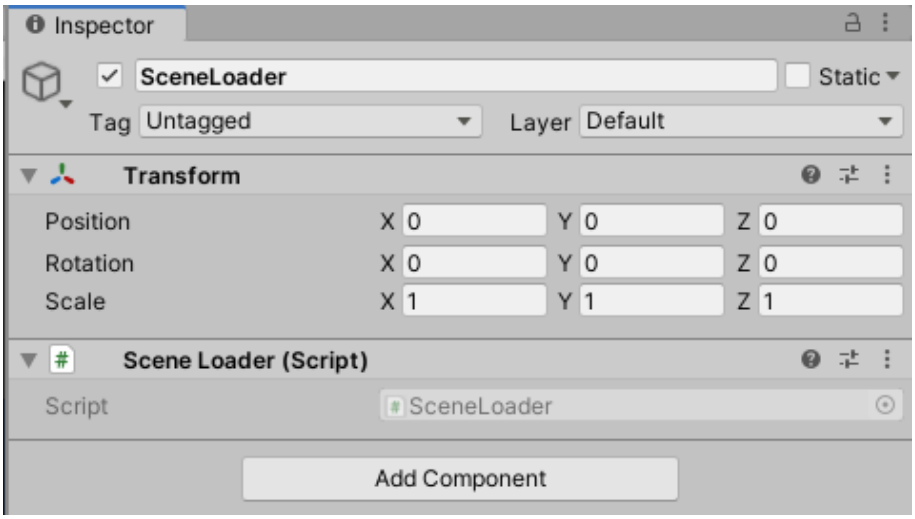


Obr. 42 Vizuál skriptu *SceneLoader.cs*.

Po návrate do Unity netreba zabudnúť pridať tento skript ako komponent objektu, ktorý sme vytvorili. Buď využijete možnosť *Add Component*, alebo skript jednoducho presuniete do okna *Inspector* (Obr. 43).

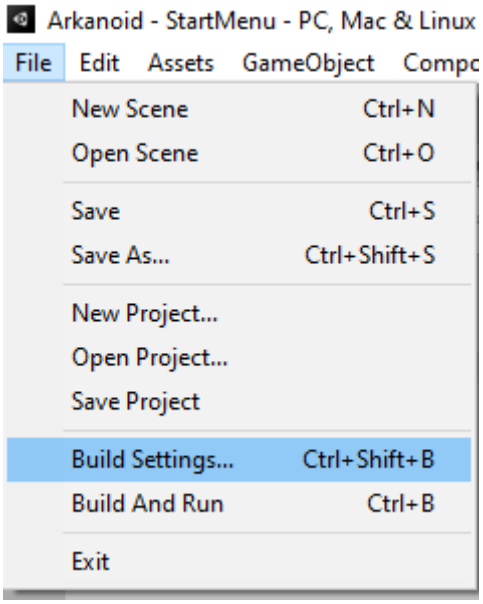


Obr. 43 Pridanie skriptu k objektu *SceneLoader*.

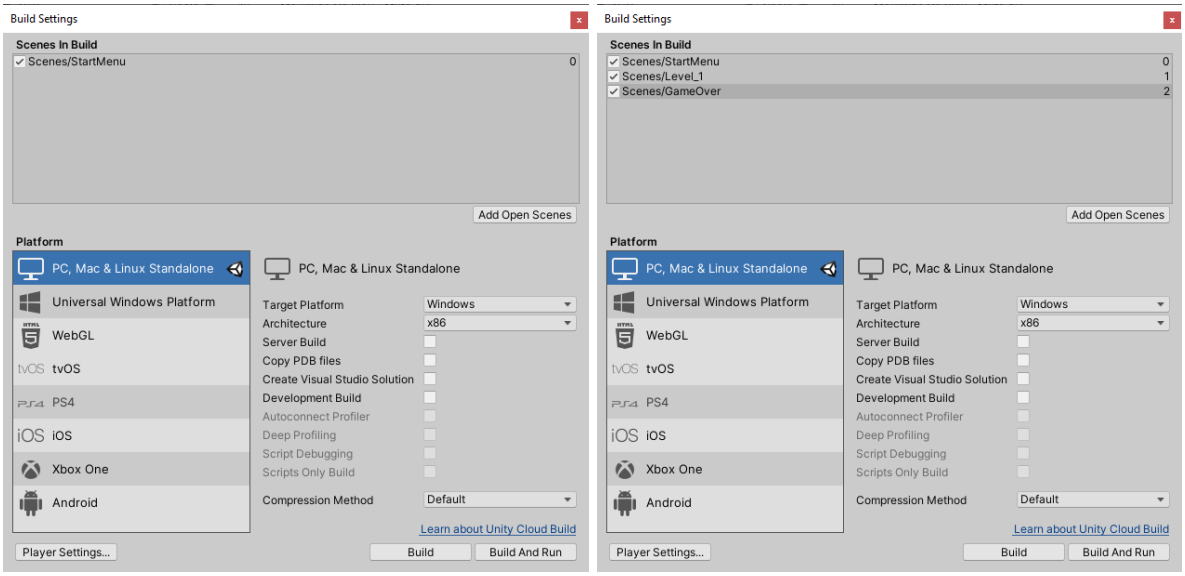


Obr. 44 Výsledok po pridaní skriptu k objektu *SceneLoader*.

Aby celé načítavanie scén fungovalo korektne, je potrebné v Unity v možnosti *File → Build Settings* (Obr. 45) pridať všetky scény s ktorými pracujeme v takom poradí ako potrebujeme, bez nutnosti exportovania výslednej hry (*Build And Run*). Pridávame ich jednoducho – presunutím scény z *Assets* (Obr. 46). Na obrázku 46 si môžeme všimnúť, že vedľa každej scény vpravo sa nachádza index scény. Po pridaní všetkých scén, stačí okno len zatvoriť.



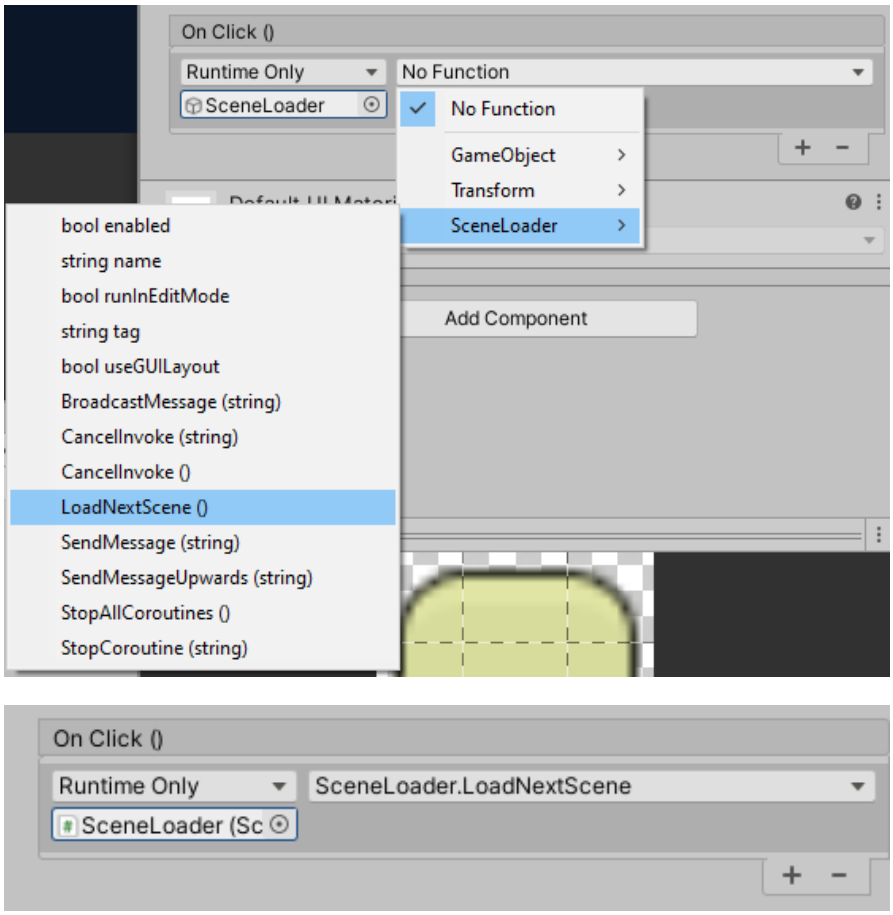
Obr. 45 Ako sprístupniť okno pre pridanie scén.



Obr. 46 Okno *Build Settings* po otvorení a po pridaní želaných scén.

Udalosť, ktorá má vyvolať prechod na ďalšiu scénu je v tomto prípade stlačenie tlačidla *StartButton*. Túto funkcionalitu zabezpečíme tak, že v okne *Inspector* tlačidla *StartButton* v možnosti *On Click* () – v tejto fáze to je prázdne (Obr. 47), pridáme objekt *SceneLoader* a využijeme vytvorenú metódu *LoadNextScene*(). Po stlačení tlačidla „+“ pridáme vytvorený objekt *SceneLoader*, napr. presunutím tohto objektu z okna *Hierarchy*, a následne musíme vybrať metódu, ktorá sa má použiť (Obr. 48).

Obr. 47 Tlačidlo ešte nemá pridanú žiadnu udalosť.



Obr. 48 Pridanie objektu *SceneLoader* a vyvolanie metódy *LoadNextScene()* spolu s výsledkom.

Tým sme zabezpečili presun z úvodnej scény na druhu scénu. Teraz potrebujeme zabezpečiť sekvenčný presun z druhej scény na tretiu, ako aj prechod na úvodnú scénu zo scény *GameOver*. Tiež potrebujeme zabezpečiť ukončenie aplikácie v prípade, ak používateľ stlačí tlačidlo *QuitButton*. Odporúčame, aby sa čitateľ pokúsil túto funkcionalitu implementovať samostatne a následne si skontroloval riešenie podľa postupu uvedeného nižšie.

Ako prvé vytvoríme v skripte *SceneLoader.cs* novú verejnú metódu, ktorá bude zabezpečovať presun na úvodnú scénu. Táto metóda je veľmi jednoduchá, pretože prvá scéna bude mať vždy index 0, preto sa stačí postarať len o načítanie scény s indexom 0 (Obr. 49):

```
public void LoadStartScene()
{
    SceneManager.LoadScene(0);
}
```

Druhú metódu pomenujeme *QuitGame()*, ktorá zabezpečí ukončenie aplikácie. K tomu poslúži metóda *Quit()* triedy *Application*. Pozor, volanie tejto metódy je ignorované v prípade spustenia hry v editore. To znamená nečakajte ukončenie aplikácie (Obr. 49).

```
public void QuitGame()
{
    Application.Quit();
}
```

SceneLoader.cs*

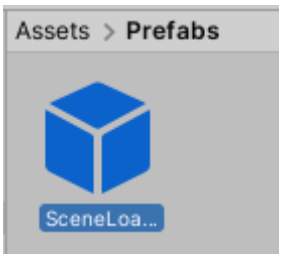
Assembly-CSharp

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement; //pridanie namespace
5
6   public class SceneLoader : MonoBehaviour
7   {
8       //verejna metoda na nacitanie scen
9       public void LoadNextScene()
10      {
11          int currentSceneIndex = SceneManager.GetActiveScene().buildIndex;
12          SceneManager.LoadScene(currentSceneIndex + 1);
13      }
14
15      public void LoadStartScene()
16      {
17          SceneManager.LoadScene(0);
18      }
19
20      public void QuitGame()
21      {
22          Application.Quit();
23      }
24  }
```

Obr. 49 Vizuál skriptu *SceneLoader.cs*.

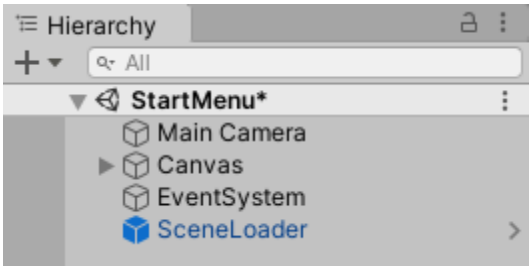
Skôr ako pridáme požadovanú funkcionalitu tlačidlám *NextButton* a *PlayAgainButton*, je vhodné z objektu *SceneLoader* vytvoriť *prefab*. *Prefab* môžeme chápať ako šablónu (*template*), ktorá ušetrí čas pri vývoji, aby ste nemuseli vytvárať nové objekty opakovane, v tomto prípade trikrát pre každú scénu. Platí, že ak potrebujete nejaký objekt spolu s jeho nastaveniami použiť v projekte opakovane, je výhodné z neho spraviť *prefab*, pretože systém *Prefab* umožňuje automaticky synchronizovať všetky jeho kópie (Unity, 2020f).

V *Assets* vytvoríme nový priečinok s názvom *Prefabs* a objekt *SceneLoader* tam jednoducho presunieme.



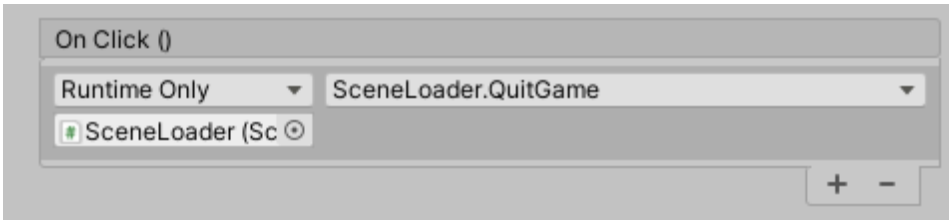
Obr. 50 Vytvorenie prefabu z objektu *SceneLoader*.

To, že sa z tohto objektu stal *prefab*, ilustruje aj zmena farby tohto objektu v okne *Hierarchy* na modrú:



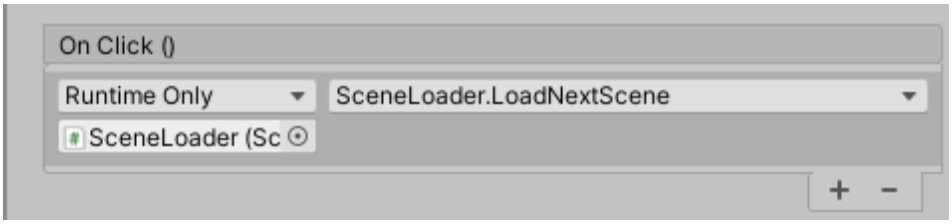
Obr. 51 Štruktúra prvej scény.

Teraz už len postačí pridať jednotlivým tlačidlám požadovanú funkcionalitu, obdobným spôsobom ako sme vysvetlili vyššie. Tlačidlu *QuitButton* pridáme volanie metódy *QuitGame()* objektu *SceneLoader*:



Obr. 52 Volanie metódy *QuitGame()* na tlačidle *QuitButton*.

Tlačidlu *NextButton* pridáme volanie metódy *LoadNextScene()*:

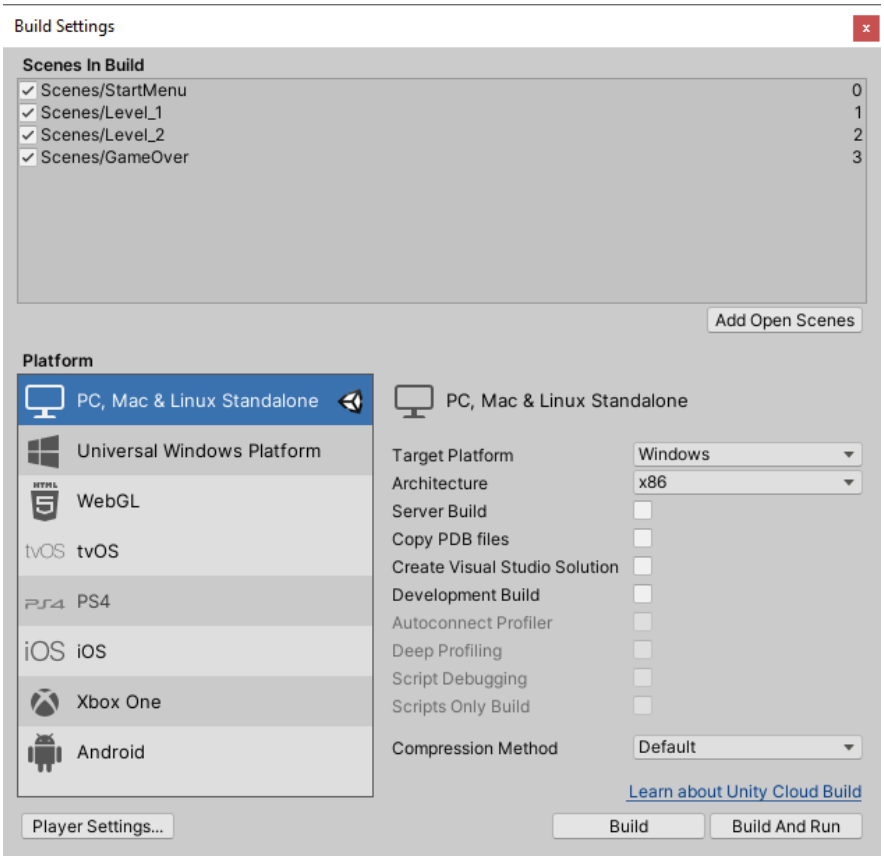


Obr. 53 Volanie metódy *LoadNextScene()* na tlačidle *NextButton*.

Tlačidlu *PlayAgainButton* pridáme volanie metódy *LoadStartScene*():

Obr. 54 Volanie metódy *LoadStartScene*() na tlačidle *PlayAgainButton*.

Spustením hry v editore pomocou *Play* tlačidla v *Toolbare* môžeme otestovať implementovanú funkcionalitu. Druhou možnosťou je, že hru exportujeme (*build*) ako výsledný projekt. Pred tým ako začneme exportovať, je vždy vhodné skontrolovať v nastaveniach *Build Settings* či sú tam všetky požadované scény a či sú v správnom poradí. Potvrdením voľby *Build*, alebo *Build And Run*, ktorý okrem exportu, spustí hneď aj hru, vyexportujeme hru pre PC s východiskovými nastaveniami (Obr. 55) (Unity, 2020g). Viac o jednotlivých nastaveniach možností exportu si povieme v kapitole 11.



Obr. 55 Okno potrebné pre exportovanie hry.

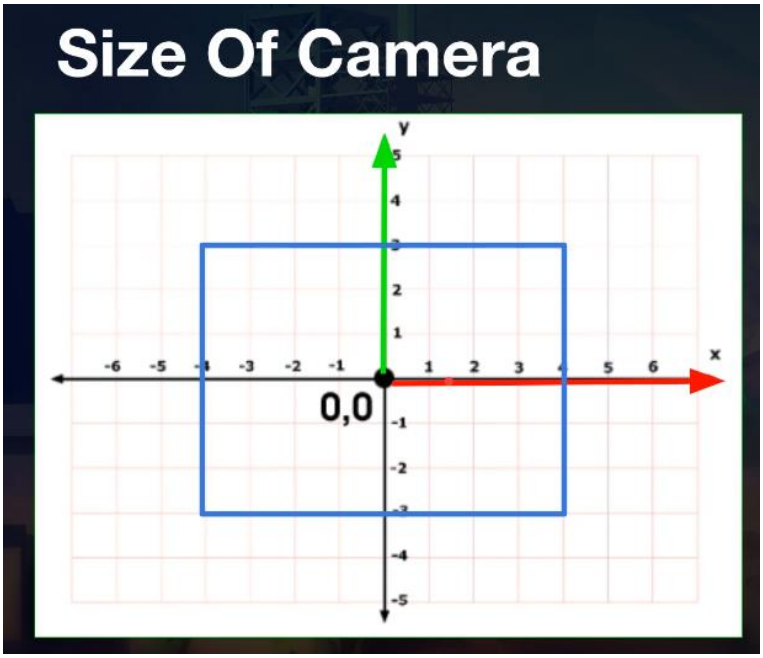
2.3 Kontrolné otázky a úlohy

1. Definujte objekt *Canvas*. Aká je jeho základná funkcionalita?
2. Čo vyjadruje termín *Aspect Ratio*?
3. Čo vyjadruje termín *UI Anchors*? V akom kontexte, prípadne s akým cieľom toto používame?
4. Aký je rozdiel medzi objektmi *Text* a *TextMeshPro*?
5. Pomocou akých vlastností vieme zabezpečiť vizuálnu zmenu správania tlačidla pri interakcii s ním?
6. Pomocou akej vlastnosti vieme zabezpečiť funkčnú zmenu správania tlačidla pri interakcii s ním?
7. Čo vyjadruje termín *Scene Index*?
8. Ako sa volá menný priestor = *namespace*, ktorý potrebujeme pre prácu so scénami?
9. Akú metódu používame na získanie aktuálnej scény?
10. Ako sa volá vlastnosť = *property*, pomocou ktorej vieme zistiť index scény?
11. Ako sa volá metóda, pomocou ktorej vieme načítať scénu? Aká je jej deklarácia?
12. Aký je rozdiel medzi verejnou = *public* a súkromnou = *private* metódou?
13. Kde určujeme, ktoré scény a v akom poradí sa budú prehrávať?
14. Čo vyjadruje termín *Prefab*? Pre aké objekty je vhodné ho používať?
15. Pomocou akej metódy vieme zatvoriť (skončiť) aplikáciu?

3 Snímanie sveta pomocou kamery

Kamera (*Camera*) je zariadenie, ktorý zachytáva a zobrazuje svet (*worldspace*) hráčovi. Prispôsobením nastavení kamery a manipuláciou s ňou môžeme urobiť prezentáciu hry jedinečnou. V scéne môžeme mať neobmedzený počet kamier. Môžu byť nastavené tak, aby sa vykresľovali v ľubovoľnom poradí, na akomkoľvek mieste na obrazovke, alebo zobrazovali len určitú časť obrazovky. Vlastnosť *Projection* hovorí o tom, či sa budú objekty vykresľovať v perspektíve alebo nie. V 2D hrách využívame nastavenie *Orthographic* (Unity, 2020h).

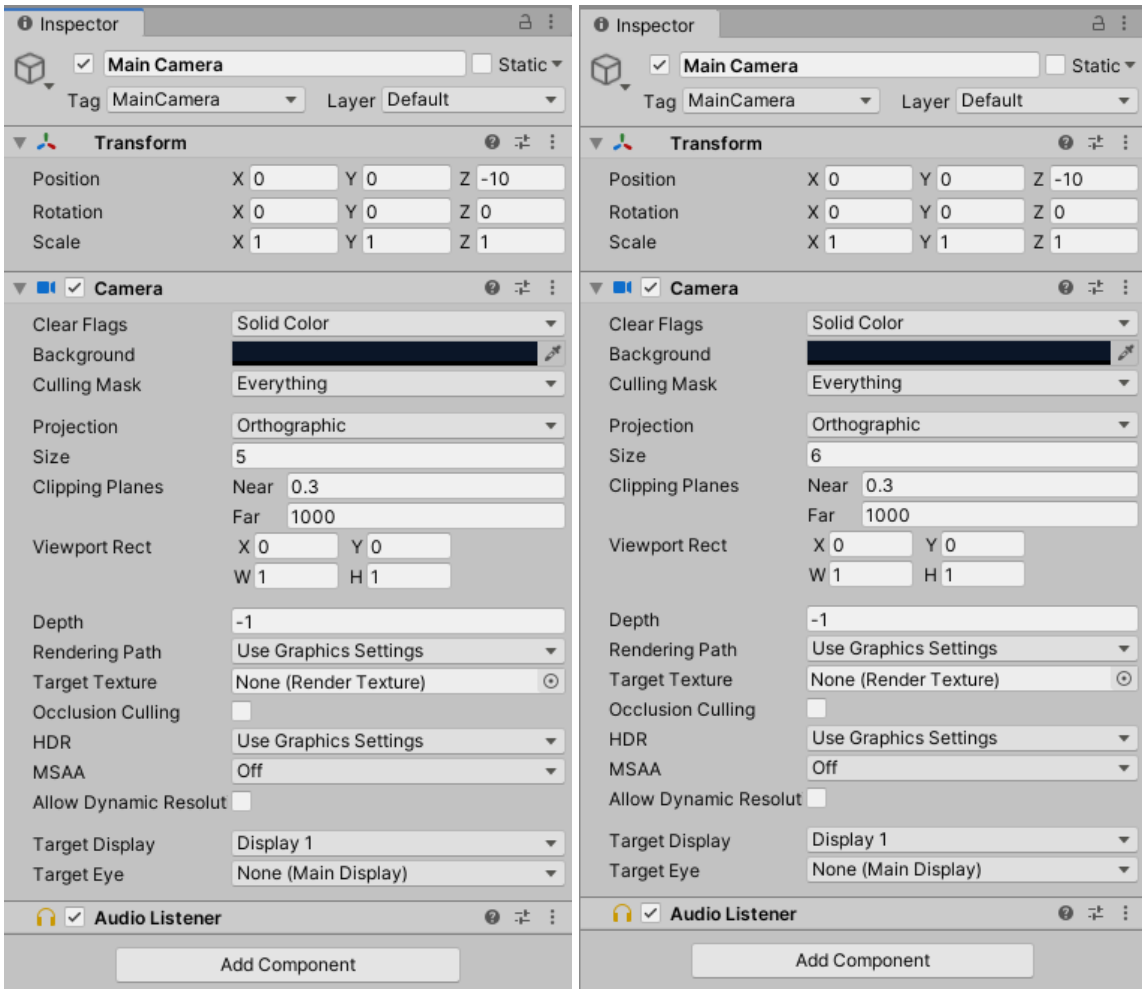
Celá plocha kamery je v 2D hrách rozdelená do jednotiek tzv. *units*. Na obrázku 56 vidíme 8 *units* horizontálne a 6 *units* vertikálne. Toto rozloženie je závislé od veľkosti kamery. Unity tieto *units* čísluje, ide o tzv. *index*. Na obrázku je vysvetlený príklad ak je *size orthographic* kamery 3. Znamená to 3 *units* vertikálne do oboch smerov, a keďže používame *Aspect Ratio* 4:3 tak na osi *x* sú 4 *units* v každom smere.



Obr. 56 Ako ovplyvňuje veľkosť kamery správanie sa vo svete (Davidson, Rick and Ben Tristen. 2019).

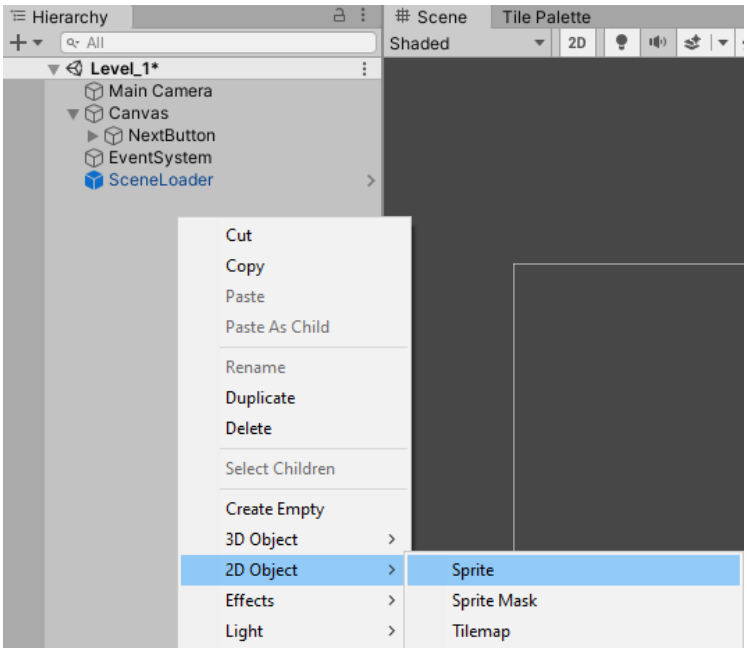
Veľkosť kamery je možné nastavovať pomocou vlastnosti *Size* v okne *Inspector*. Momentálne je veľkosť kamery v scéne *Level_1* nastavená na 5, my jej veľkosť zmeníme na 6, čo znamená 12 *units* vertikálne a 16 *units* horizontálne.

Pozn. autora: Vo všeobecnosti platí, ak je číslo kamery veľmi veľké, tak používaný *assets* je príliš malý, ak malé tak je *assets* príliš veľký.



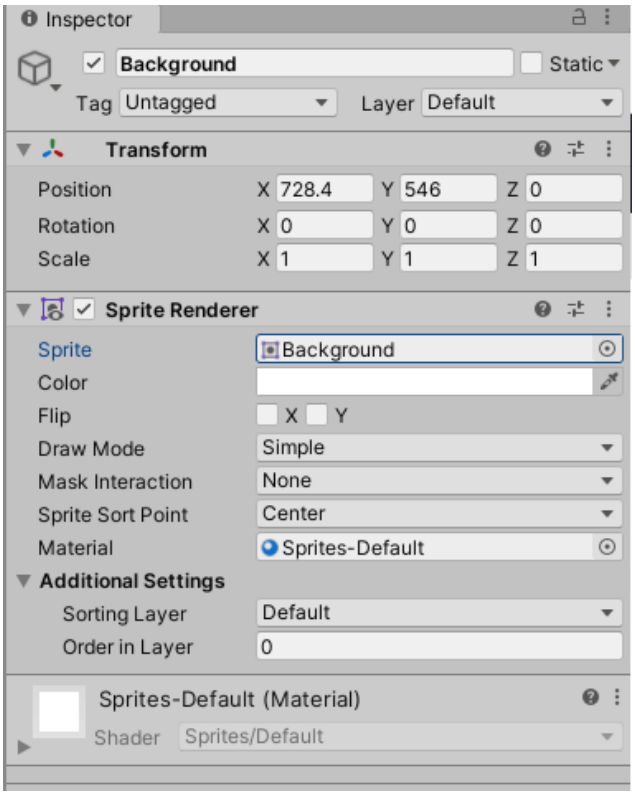
Obr. 57 Pôvodné a upravené nastavenia kamery.

Teraz pridáme do scény *Level_1* pozadie. Urobíme to tak, že v okne *Hierarchy* pridáme nový 2D objekt – *Sprite* a pomenujeme ho *Background* (Obr. 58).



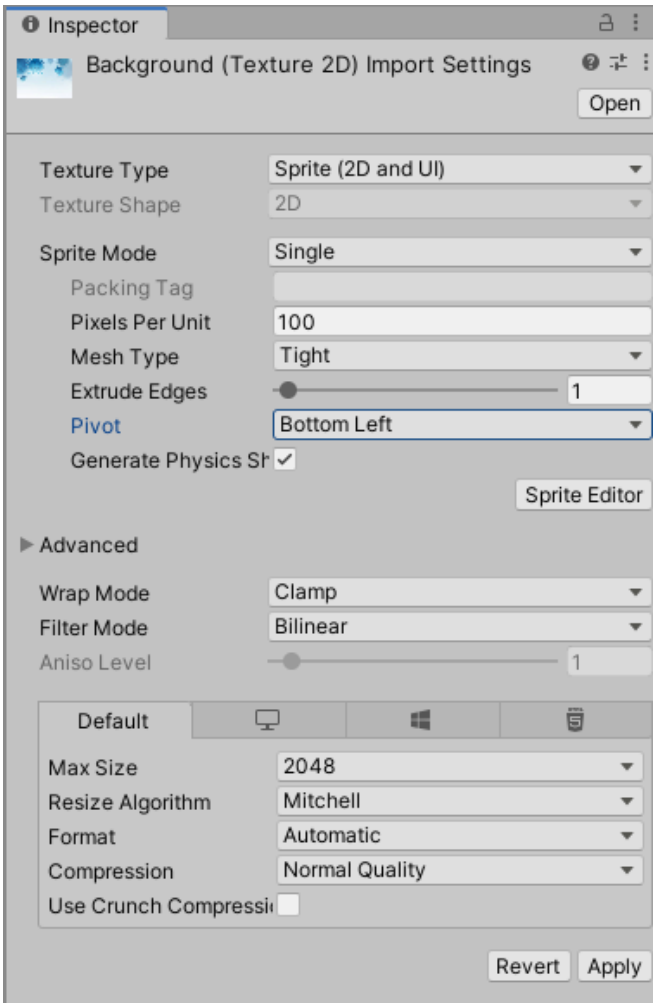
Obr. 58 Pridanie 2D objektu – *Sprite*.

Skôr ako začneme nastavovať vlastnosti tohto objektu, tak si v Assets vytvoríme priečinok s názvom *Sprites*, kde vložíme obrázok pozadia, ktorý chceme použiť. Potom pomocou vlastnosti *Sprite* nastavíme pozadie na požadovaný obrázok (Obr. 59).



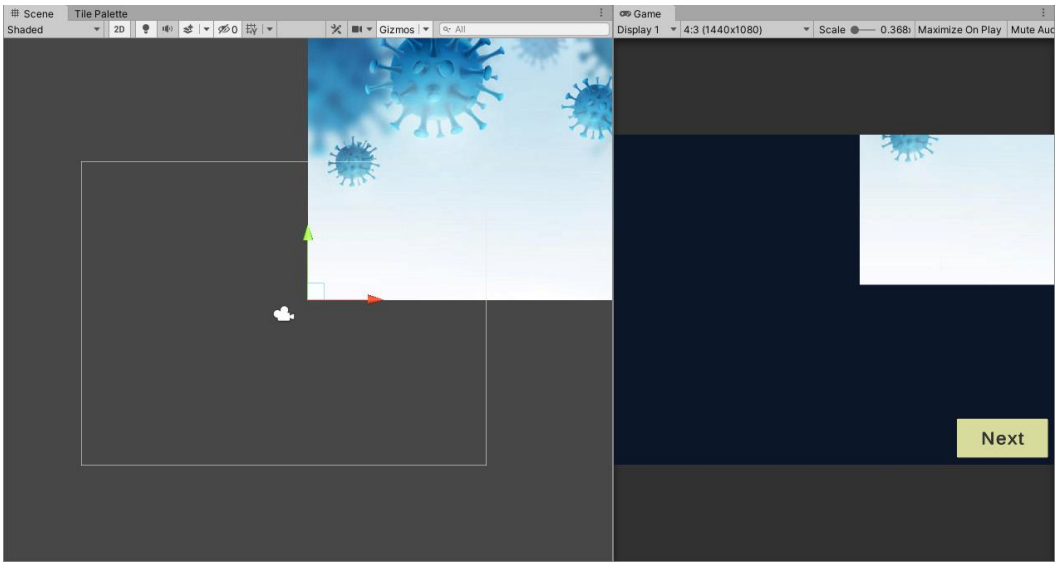
Obr. 59 Nastavenie zdrojového súboru pre objekt *Sprite*.

Vo vlastnostiach obrázka zmeníme nastavenie vlastnosti *Pivot* na *Bottom Left*. Pri vykonávaní tejto zmeny je nutné mať vybratý *sprite* z priečinku *Sprites*. Zmena pivotu z možnosti *Center* na *Bottom Left* nám uľahčí prácu s ním, konkrétne jeho umiestnenie vzhľadom na svet (*worldspace*). Pozor, zmeny treba aplikovať pomocou tlačidla *Apply*, inak sa neprejavia (Obr. 60).



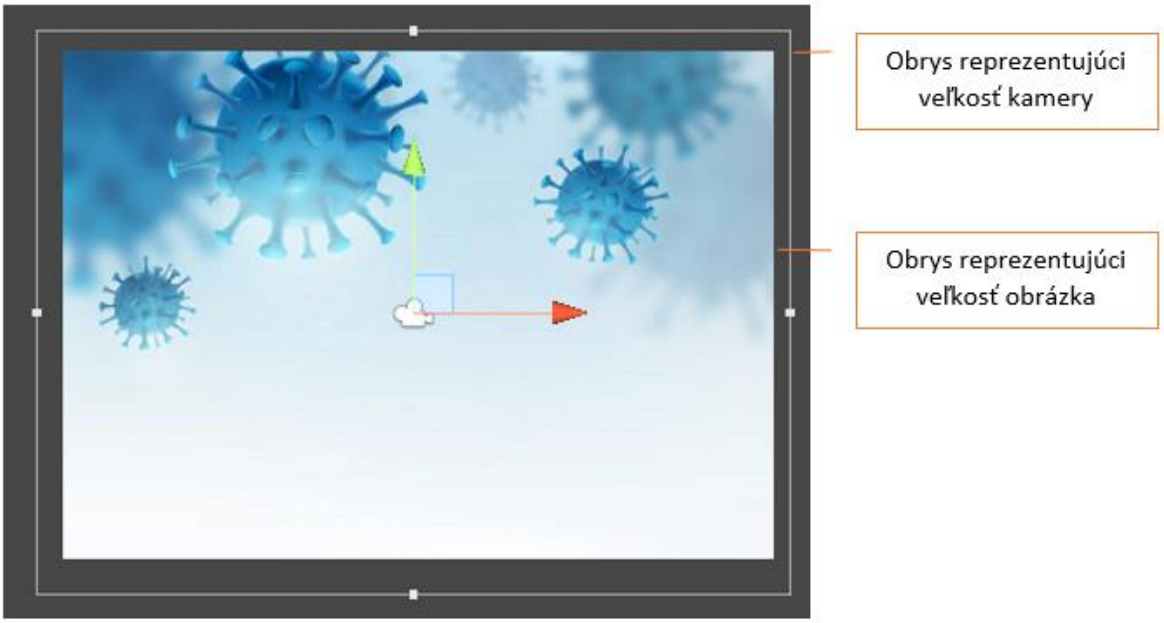
Obr. 60Zmena pivotu obrázka.

Ako môžeme vidieť na obrázku 61, kamera nezachytáva presne naše pozadie.



Obr. 61 Nastavenie kamery a pozadia.

Pozíciu kamery zmeníme tak, že ju umiestnime približne do stredu obrázka. Zistíme, že kamera je o niečo väčšia ako je obrázok pozadia (Obr. 62).

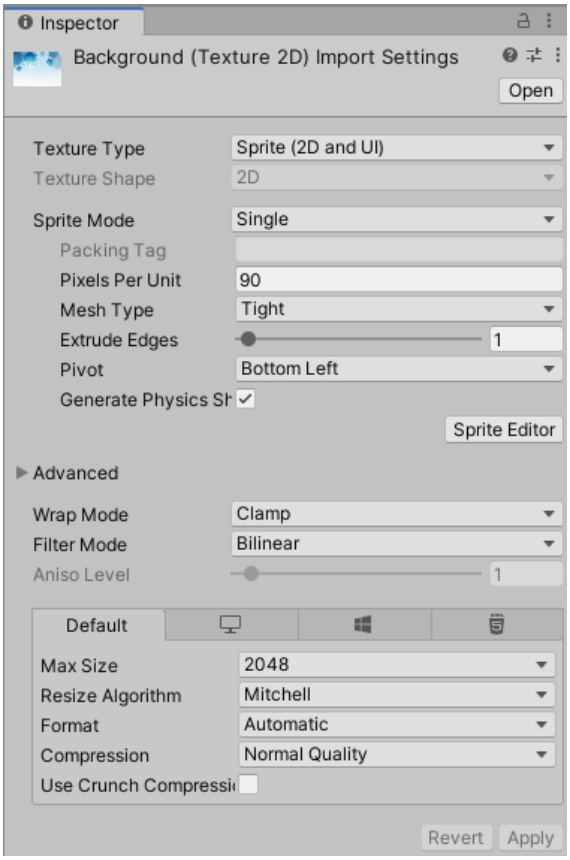


Obr. 62 Veľkosť kamery voči obrázku pozadia.

Obrázok potrebujeme prispôsobiť veľkosti kamery, čo urobíme tak, že vhodne zmeníme veľkosť obrázka vzhľadom na veľkosť kamery. Ak veľkosť obrázka vo vertikálnom smere je 1080 (Obr. 63) a túto hodnotu vydelíme číslom 12, pretože máme 12 *units* vo vertikálnom smere (vzhľadom na použitú veľkosť kamery a *Aspect Ratio*), dostaneme hodnotu 90.

Obr. 63 Skutočná veľkosť obrázka.

Na túto hodnotu je potrebné nastaviť vlastnosť obrázka *Pixels Per Unit* a aplikovať zmeny pomocou tlačidla *Apply*.

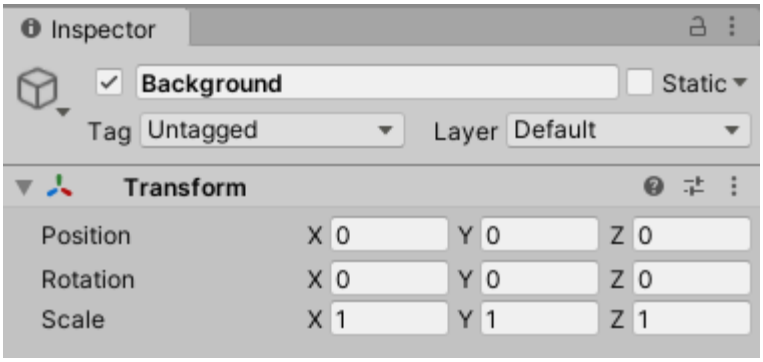


Obr. 64 Modifikácia vlastnosti – *Pixel Per Unit*.

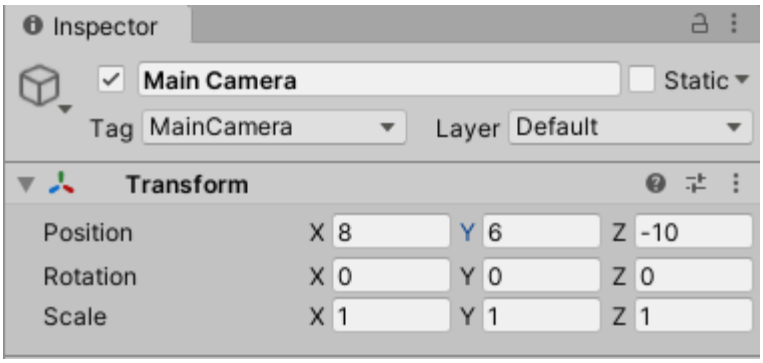
Výsledkom je, že obrázok je teraz o niečo väčší než kamera. Ak nám to vyhovuje, tak nie je nutné mať veľkosť obrázka presne rovnakú ako je veľkosť kamery (Obr. 65).

Obr. 65 Veľkosť kamery voči zmenenému obrázku.

Aby kamera presne snímala obrázok, tak zmeníme pozíciu obrázka v smere osi *x* a *y* na 0 (Obr. 66). Pozíciu kamery nastavíme presne do stredu, t. j. v smere osi *x* na hodnotu 8 a v smere osi *y* na hodnotu 6 (vzhľadom na veľkosť kamery = 6 a *Aspect Ratio* 4:3 sú práve toto súradnice stredu) (Obr. 67).

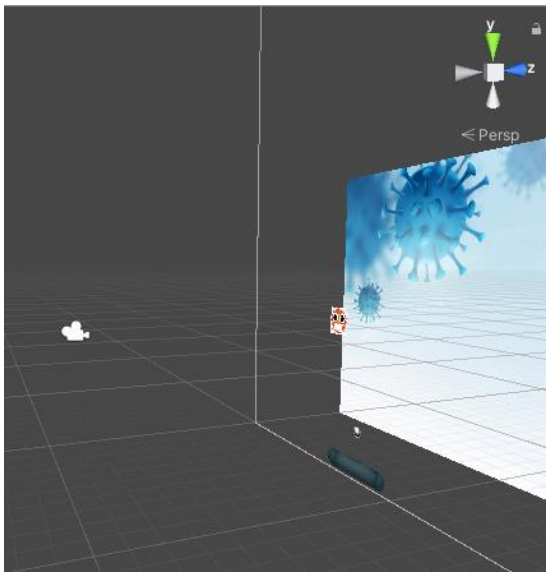


Obr. 66 Nastavenia pozície obrázka pozadia.



Obr. 67 Nastavenia pozície objektu *Main Camera*.

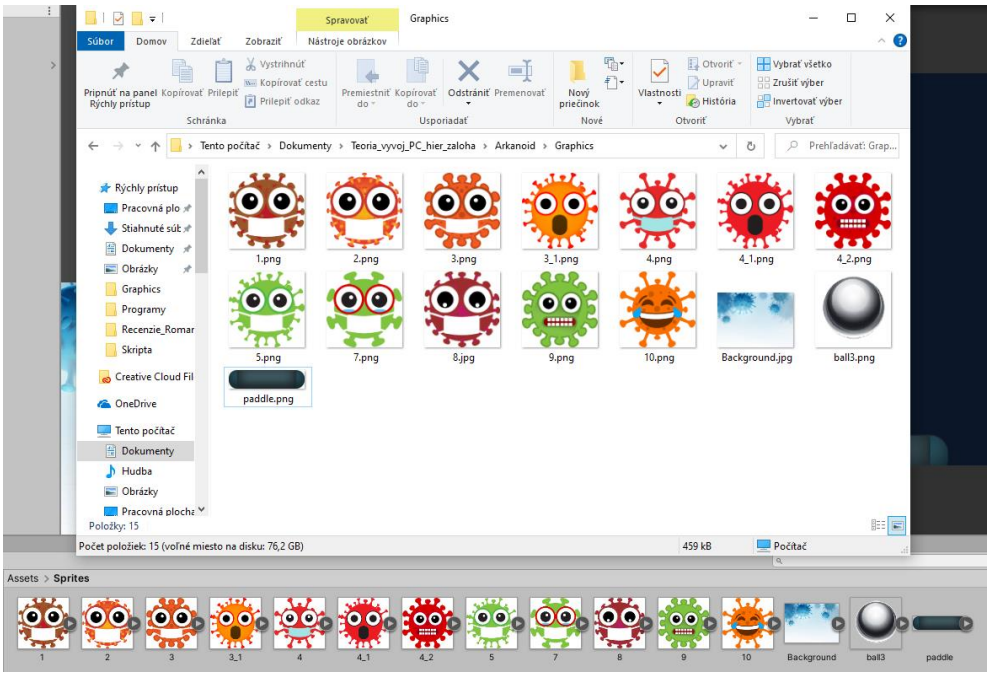
Pozn. autora: Ja zaujímavé si všimnúť, že pozícia kamery v smere osi z je nastavená na hodnotu –10, čo znamená, že kamera je mierne vysunutá nad ostatnými objektmi v scéne. Ak máme všetky objekty umiestnené na pozícii v smere osi z rovnej 0, tak sa môže stať, že niektoré objekty prekryjú iné. Kým ešte nepracujete s vrstvami, pomocou ktorých sa dá tento problém riešiť, je riešením upravovať zobrazenie objektu v smere osi z. Obrázok 68 ilustruje tieto rozdielne nastavenia v smere osi z v prípade použitia viacerých objektov v scéne. Pozadie sa nachádza za prvkami reprezentujúcimi blok, padlo a loptičku. Toto perspektívne zobrazenie je možné zobraziť pomocou tlačidla „2D“ nachádzajúceho sa v okne *Scene view* (Obr. 69).



Obr. 68 Zobrazenie scény v perspektíve.

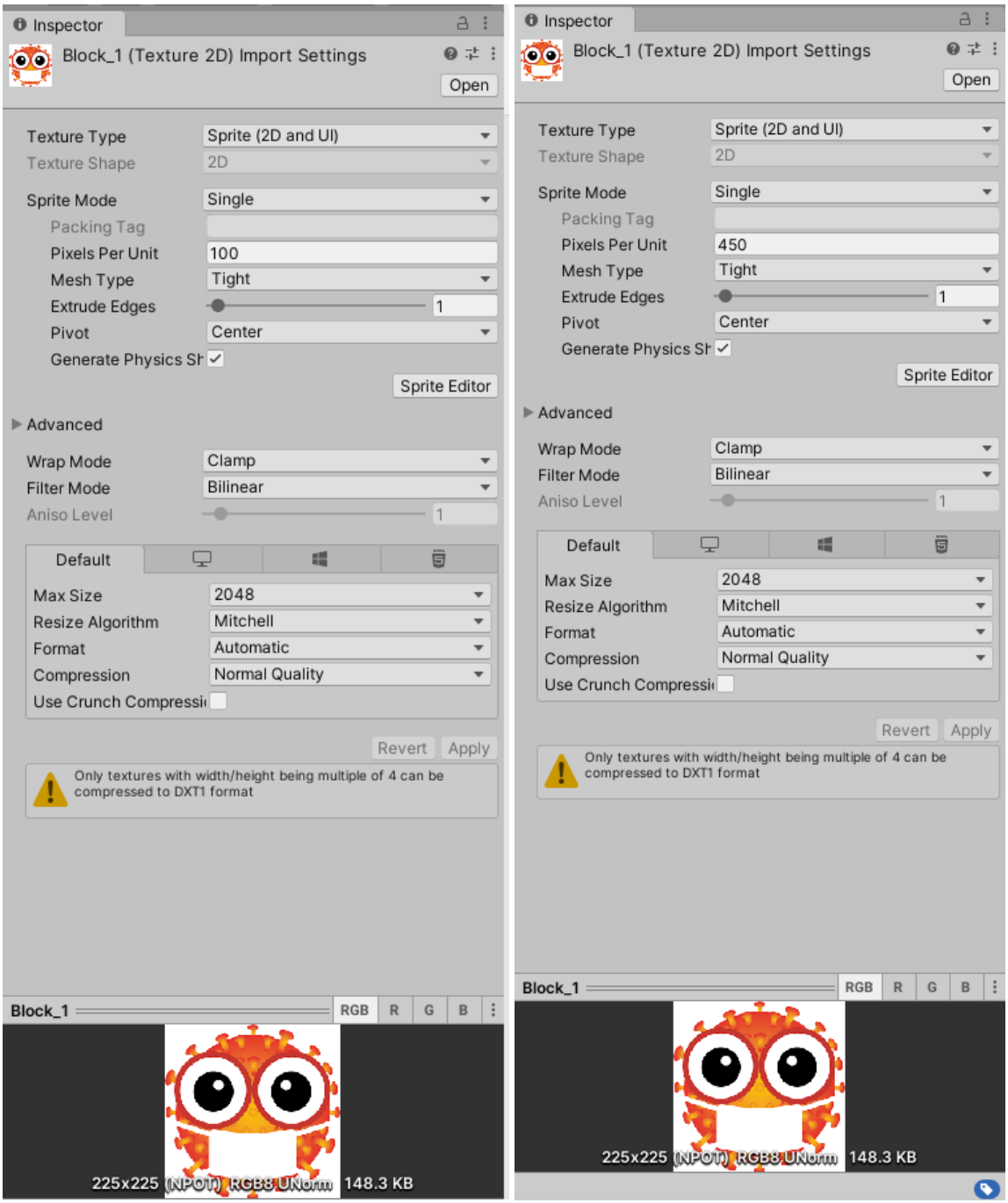
Obr. 69 Tlačidlo 2D pre zobrazenie scény v perspektíve.

Do scény pridáme ďalšie objekty, reprezentujúce základné elementy hry – blok, padlo a loptičku. Vhodné grafické reprezentácie týchto objektov vložíme do priečinku *Sprites*.



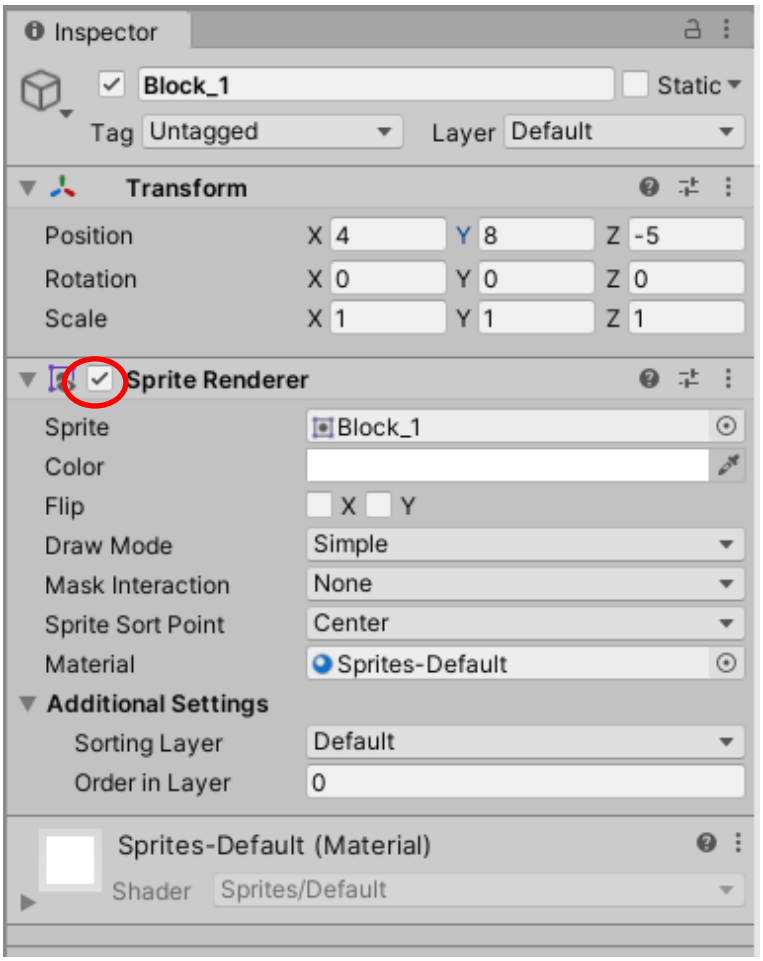
Obr. 70 Vloženie grafického aparátu do projektu.

Ak je reálna veľkosť obrázka, ktorý predstavuje blok 225 x 225, pri nastavení *Pixel Per Unit* na hodnotu 225, bude možné v horizontálnom smere umiestniť maximálne 16 blokov, pretože svet predstavuje 16 *units*. Pre hrateľnosť hry to je dosť málo, preto v prípade zdvojnásobenia hodnoty *Pixel Per Unit*, zdvojnásobíme aj tento počet. Nezabudnite všetky zmeny aplikovať pomocou tlačidla *Apply*.



Obr. 71 Pôvodné a zmenené nastavenia obrázka reprezentujúceho blok.

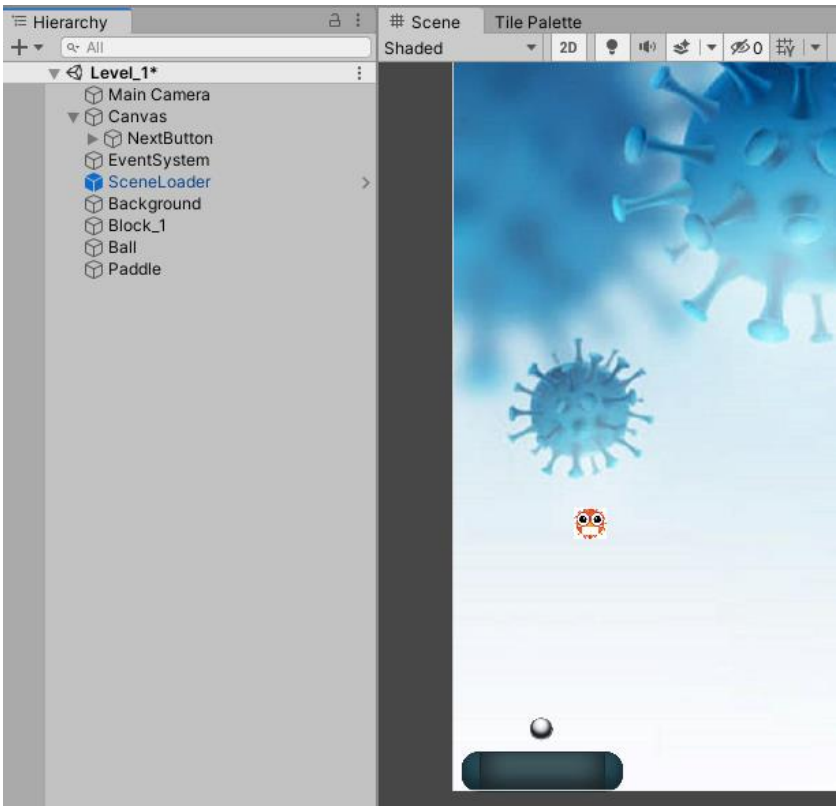
Objekt pridáme do scény jednoduchým presunutím tohto *spritu* buď priamo do okna *Hierarchy*, alebo do scény. Vzniknutý objekt premenujeme na *Block_1* a zmeníme v okne *Inspector* v komponente *Transform* pozíciu v smere osi *z* napr. na −5 (Obr. 72), aby bloky boli na inej úrovni ako pozadie, ktoré má $z = 0$. Tento objekt obsahuje aj komponent *Sprite Renderer*, kde napríklad môžeme vidieť, že ako zdrojový obrázok (vlastnosť *Sprite*) je použitý obrázok s názvom *Block_1*.



Obr. 72 Nastavenie vlastností objektu *Block_1*.

Pozn. autora: Odškrtnutím *checkboxu* v komponente *Sprite Renderer* vlastne deaktivujeme tento komponent, čo bude mať za následok, že objekt v scéne neuvidíme, aj keď sa tam nachádza.

Obdobne pridáme do scény aj loptičku – objekt premenujeme na *Ball* a *Pixel Per Unit* zmeníme na 280. Padlo premenujeme na *Paddle* a *Pixel Per Unit* zmeníme na 200. Hodnoty upravujeme v závislosti od ich reálnej veľkosti vzhľadom na veľkosť použitej kamery. Obidva objekty umiestnime na tú istú úroveň ako blok, t. j. ich pozíciu v smere osi *z* nastavíme na hodnotu –5. Štruktúru a vizuálnu podobu scény Level_1 ilustruje obrázok 73.



Obr. 73 Vizuál a štruktúra scény *Level_1*.

3.1 Kontrolné otázky a úlohy

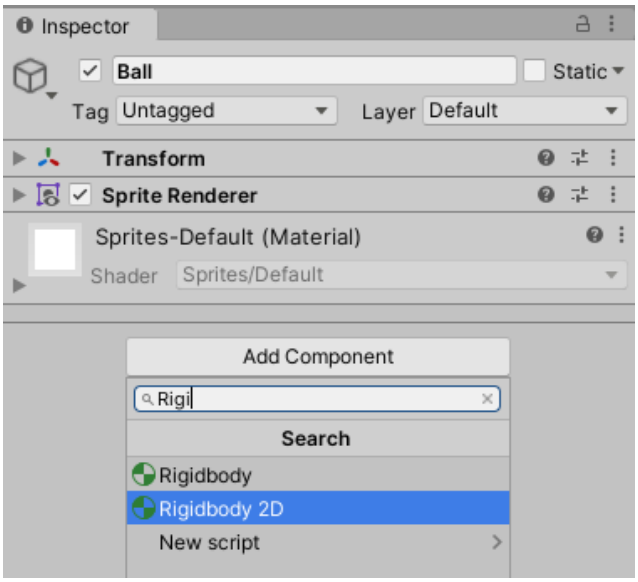
1. Čo vyjadruje termín *units*?
2. Ako sa *units* mení v závislosti od veľkosti kamery?
3. Čo vyjadruje termín *pivot*?
4. Z akého dôvodu zasahujeme aj do nastavení pozície v smere osi *z* keďže sa venujeme vývoju 2D hry?
5. Čo ovplyvňuje a ako funguje vlastnosť obrázka (*spritu*) *Pixel Per Unit*?
6. Experimentuje s nastavením vlastnosti *Size* pre kameru, a vypočítajte si *units* pre daný svet vzhľadom na použitý *Aspect Ratio*.
7. V prípade použitia vlastných grafických elementov, umiestnite tieto do scény a vhodne prispôsobte ich veľkosť vzhľadom na svet.

4 Kolízie objektov a ich možnosti

Aby sme mohli sledovať vzájomné kolízie pohybujúcich sa objektov a príslušne na ne reagovať, je dôležité, aby tieto objekty mali pridaný komponent *Rigidbody 2D*. *Collider* je možné pridať aj objektom, ktoré nemajú komponent *Rigidbody 2D*, vtedy však hovoríme o statických *collideroch*, ktoré sa najčastejšie používajú pre objekty ako podlaha, stena a iné nepohyblivé objekty v scéne. V závislosti od nastavení *collideru* môžu objekty medzi sebou interagovať. Rozlišujeme tri konfigurácie *colliderov*:

- ***Static Collider*** – predstavuje objekt, ktorý má *collider*, ale nemá *Rigidbody*. Väčšinou sa používajú pre statické objekty, ktoré nemenia svoju pozíciu a nepohybujú sa. Iné objekty môžu kolidovať s nimi, ale nemôžu ich presunúť.
- ***Rigidbody Collider*** – predstavuje objekt s *colliderom* a normálnym (*non–Kinematic*) *Rigidbody*. Takýto objekt je schopný aplikovať na seba fyzikálne zákonitosti, ktoré je možné nastavovať aj pomocou skriptu.
- ***Kinematic Rigidbody Collider*** – predstavuje objekt s *colliderom* a *Kinematic Rigidbody*. Tento objekt má vlastnosť *isKinematic* nastavenú ako aktívnu. Takýto objekt je možné vzhľadom na kolíziu pohybovať, ale aj skryť. Ale nie je možné na neho aplikovať fyzikálne sily, ako na *Rigidbody Collider* (Unity, 2020i).

Objektu *Ball* pridáme komponent *Rigidbody 2D*, pomocou možnosti *Add Component* v okne *Inspector* (Obr. 74).



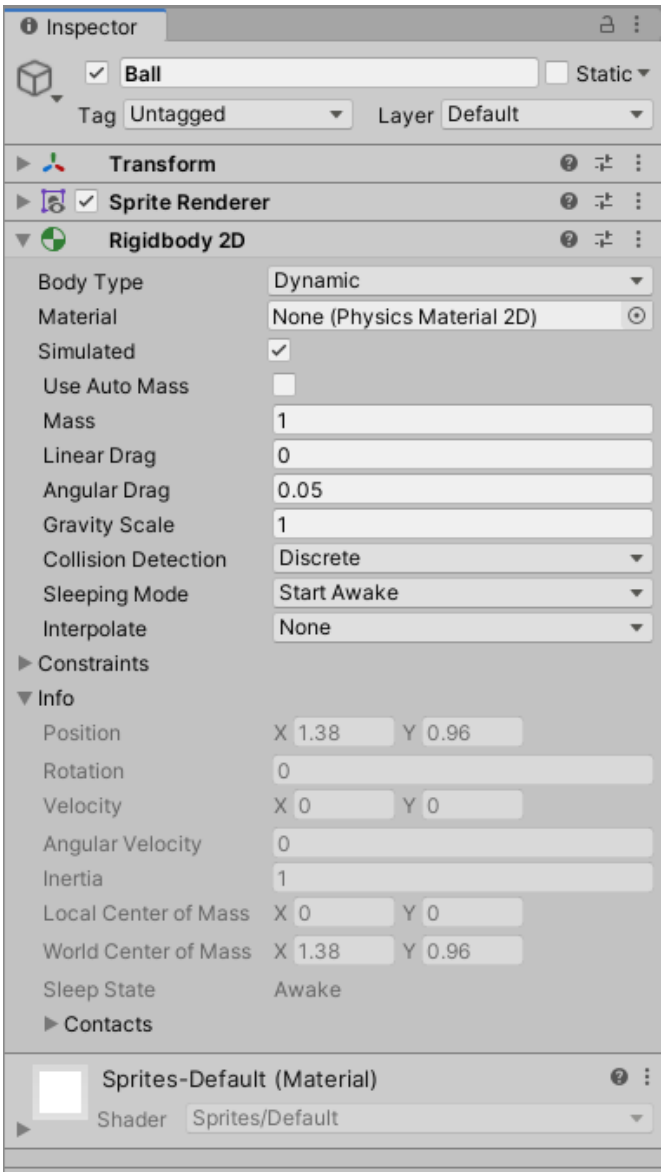
Obr. 74 Pridanie komponentu *Rigidbody 2D*.

Vlastnosť *Body Type* (Obr. 75) ovplyvňuje pohyb (*movement*) objektu, konkrétne pozíciu (*position*) a rotáciu (*rotation*), ako aj interakciu pri kolízii (*collider interaction*). Existujú tri možnosti nastavenia:

- ***Dynamic*** – objekt s týmto natavením môže interagovať s každým iným objektom, ktorý má *Rigidbody*, ako aj so všetkými ostatnými objektmi. Ide o najinteraktívnejšiu formu, preto je tento typ aj východiskovým nastavením. Avšak jeho použite sa odráža na výkone (*performance-expensive body type*). O tom, ako bude objekt reagovať rozhoduje *physic engine* ktorý, aplikuje na objekt sily a gravitáciu, čo je možné robiť dynamicky pomocou skriptu, alebo priamo pomocou nastavení vlastností *Collisions* a *Gravity*. Tento typ využijeme pre objekt *Ball*.
- ***Kinematic*** – toto nastavenie umožňuje tiež aplikovať pohyb na objekt, ale tento nie je ovplyvňovaný silami a gravitáciou, ako to bolo pri type *Dynamic*. Z tohto dôvodu je tento proces rýchlejší a aj menej náročný na zdroje. Objekt s takýmto typom môže interagovať len s objektmi *Dynamic Rigidbody 2D*. Tento typ využijeme pre objekt *Paddle*, keďže jeho pohyb chceme mať plne pod kontrolu (padlo budeme ovládať pomocou pohybu myši).

- **Static** – používa sa pre nepohybujúce, resp. nepohyblivé objekty. Nie je ovplyvňovaný silami a ani gravitáciou. *Static Rigidbody 2D* interaguje len s *Dynamic Rigidbody 2D* (Unity, 2020j).

Ako sme uviedli, pre objekt *Ball* použijeme *Dynamic Body Type*:

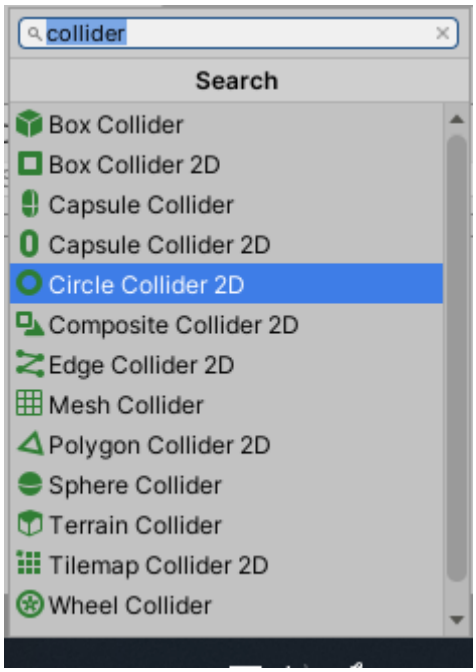


Obr. 75 *Body Type* pre objekt *Ball*.

4.1 Pridanie *collideru* objektu *Ball* a *Block_1*

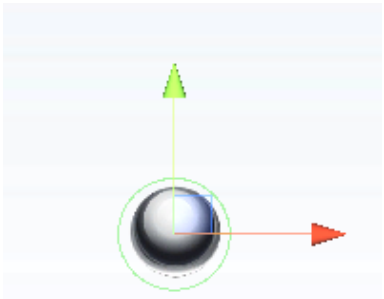
Objektu *Ball* pridáme ďalší komponent a to *Circle Collider*. Z ponuky na obrázku 76 môžeme vidieť, že k dispozícii máme viacero typov *colliderov*. *Collider* v podstate

definuje tvar objektu pre účely detekcie kolízií. Collider je neviditeľný, z tohto dôvodu nepotrebuje mať presný tvar ako je objekt, t. j. nemusí byť totožný s *meshom* objektu. Zvyčajne sa ako *collider* používajú jednoduché geometrické útvary. V rámci jedného objektu môžeme definovať viacero *colliderov*, vtedy hovoríme o tzv. *compound colliders* (Unity, 2020i).



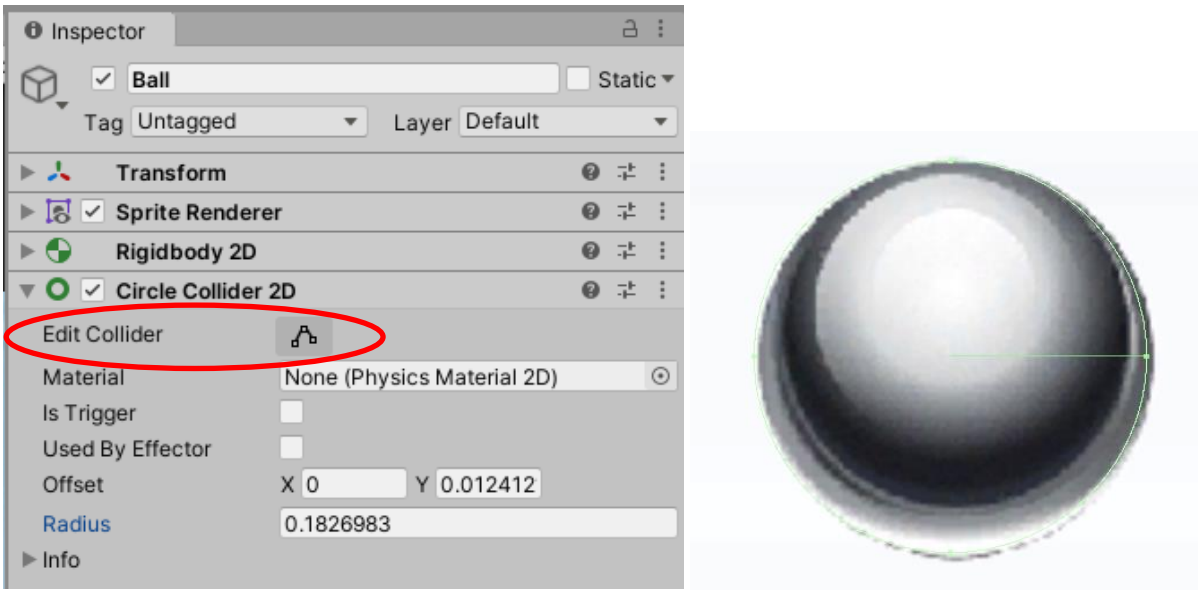
Obr. 76 Pridanie komponentu *Circle Collider 2D*.

Po pridaní *collideru* je možné vidieť zelený kruh okolo objektu *Ball* – to je *collider* (Obr. 77).



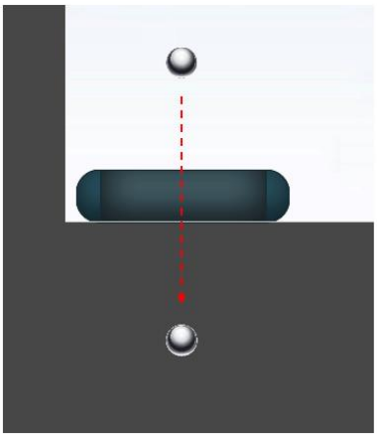
Obr. 77 *Collider* na objekte *Ball*.

Zmenu veľkosti *collideru*, čo najpresnejšie k veľkosti objektu je možné urobiť pomocou *Edit Collider* a úchytných bodov na *collidery*, alebo menej presným spôsobom a to nastavením vlastnosti *Radius* na hodnotu cca 0,18 (Obr. 78).



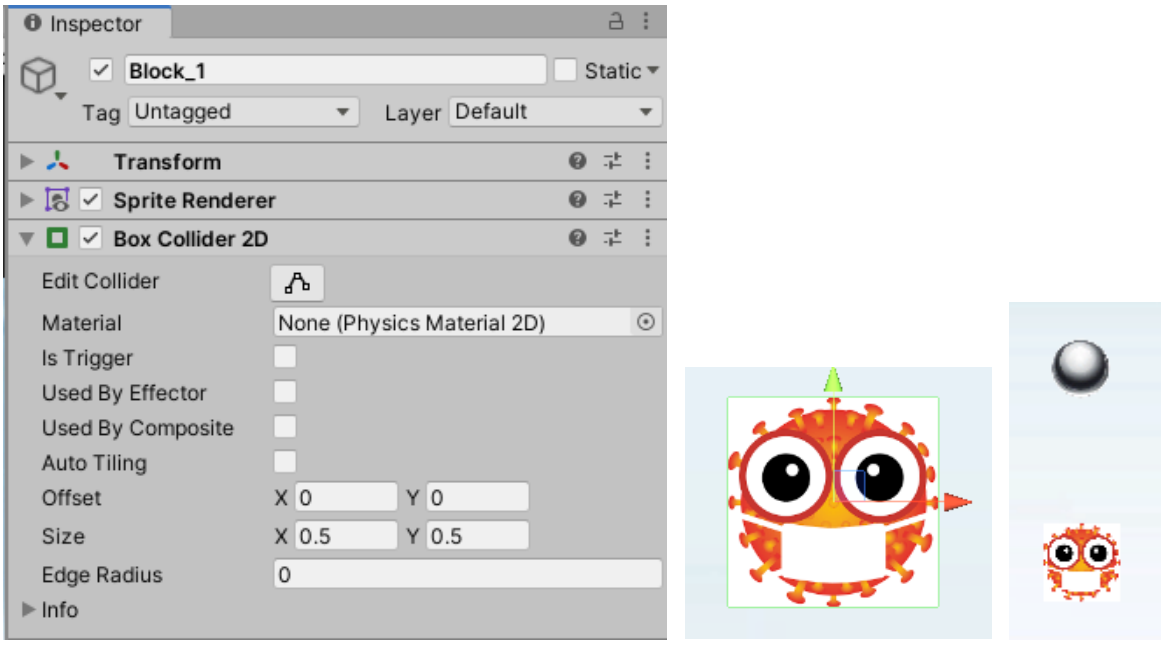
Obr. 78 Prispôsobenie veľkosti *Circle Collider* objektu *Ball*.

Ak teraz otestujeme správanie sa hry pomocou tlačidla *Play* nachádzajúceho sa v *Toolbare*, tak loptička začne padať v dôsledku aplikovanej gravitácii od miesta jej umiestnenia až mimo scénu (Obr. 79).



Obr. 79 Zachytenie padajúcej loptičky pod vplyvom pôsobenia gravitácie.

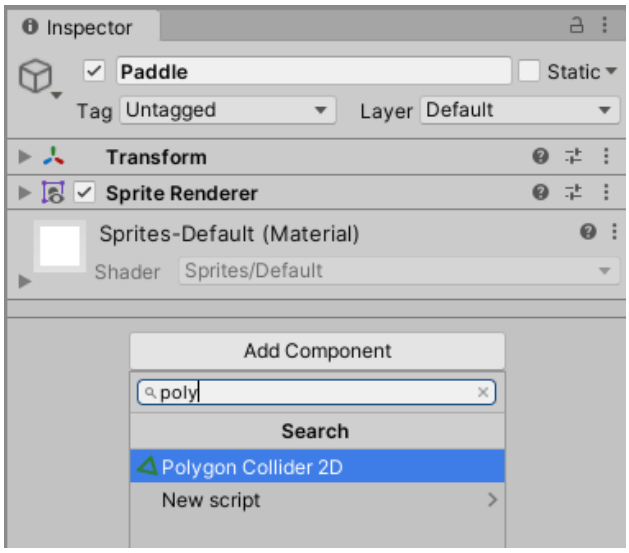
Aby sme mohli sledovať ako reagujú jednotlivé *collidery* pri kolízii objektov, doplníme *BoxCollider 2D* objektu *Block_1* (Obr. 80). Tento objekt umiestnime pod objekt *Ball*. Opätovným otestovaním hry, si môžeme všimnúť, že loptička sa tentokrát zastaví na bloku.



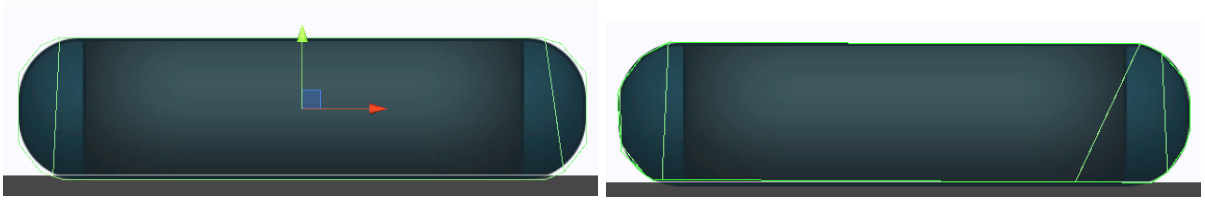
Obr. 80 Pridanie *Box Collideru 2D* objektu *Block_1* a umiestnenie objektu *Ball* a *Block_1* v scéne.

4.2 Pridanie *collideru* objektu *Paddle*

V prípade, ak by sme pracovali na vývoji 3D hry, môžeme objektu, pri ktorom chceme aby *collider* presne kopíroval jeho tvar, použiť *Mesh Collider*. V 2D priestore pre tento istý účel používame *Polygon Collider* (Obr. 81). Pomocou *Edit Collider* je možné doupravovať jeho tvar podľa objektu, na ktorý je aplikovaný (Obr. 82). Treba si však uvedomiť, že tento typ *collideru* je náročnejší na spracovanie, ako primitívne typy *colliderov*, ktoré sme použili pri objektoch *Ball* a *Block_1*.



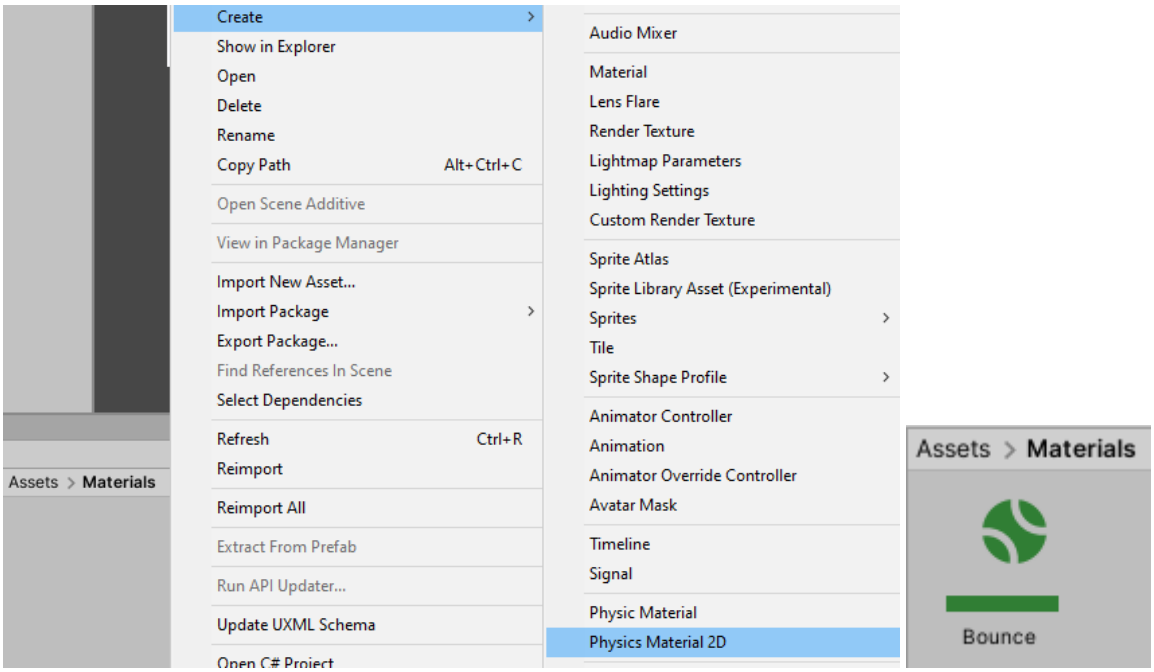
Obr. 81 Pridanie *Pollygon Collideru* objektu *Paddle*.



Obr. 82 Vizuál *collideru* pred a po aplikovaní zmien pomocou *Edit Collider*.

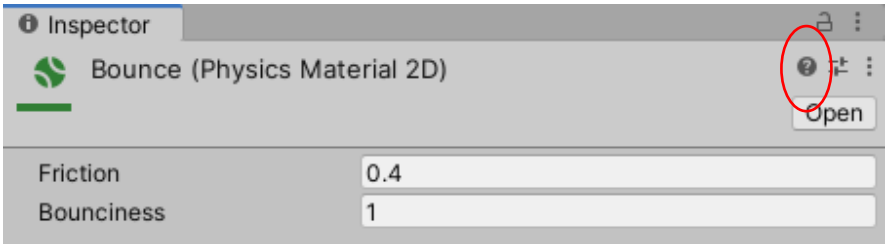
4.3 Pridanie materiálu objektu *Ball*

Ak potrebujeme v Unity zabezpečiť fyzikálne správanie sa objektov pri ich kolízii, docielime to aplikovaním materiálu použitím komponentu *Physics Material 2D*. Vďaka nemu je možné simulovať materiál ako ľad, gumu a pod., t. j. napríklad ľad je klzký, zatiaľ čo guma ponúka veľké trenie (*friction*) a bude sa odrážať (*bounce*) (Unity, 2020k). Cieľom je zabezpečiť, aby sa loptička odrazila v prípade kolízie s inými objektmi. Pomocou kontextovej ponuky v Assets vytvoríme nový priečinok s názvom *Materials*, kde vytvoríme nový materiál s názvom *Bounce*.



Obr. 83 Vytvorenie *Physics Material 2D*.

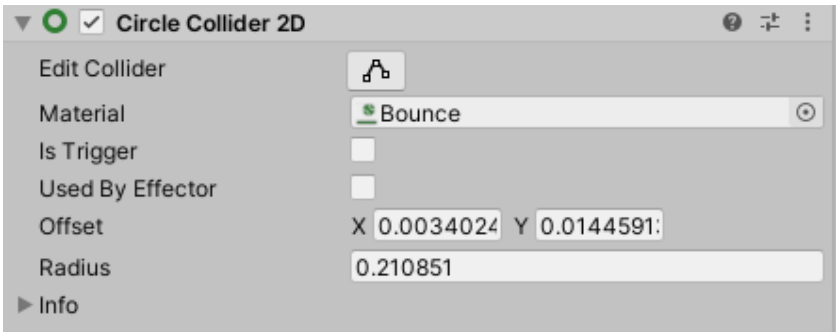
Pomocou okna *Inspector* nastavíme požadované vlastnosti. Vlastnosť *friction* nie je v tomto prípade pre nás až taká dôležitá, ako vlastnosť *Bounciness*, ktorá sa nastavuje v rozpätí 0 ÷ 1, pričom nastavenie na hodnotu 0 znamená, že sa neodráža, nastavenie na 1, že sa odráža perfektne, bez straty energie.



Obr. 84 Nastavenie materiálu *Bounce*.

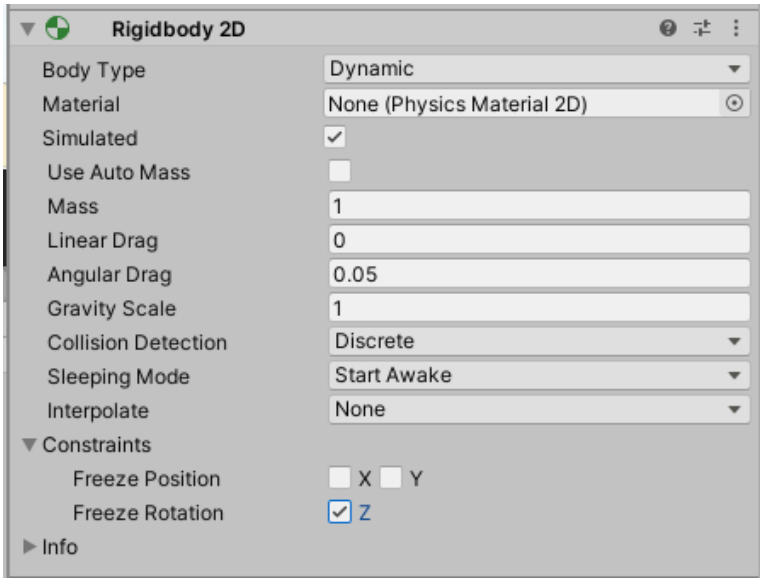
Pozn. autora: Pomocou tlačidla s ikonou „?“ je možné vyvolať Unity dokumentáciu o konkrétnom komponente s ktorým pracujeme.

Na objekt sa takýto materiál aplikuje veľmi jednoducho, stačí potiahnuť do komponentu *Circle Collider 2D* do vlastnosti *Material* vytvorený materiál *Bounce*.



Obr. 85 Pridanie materiálu *Bounce* objektu *Ball*.

Pri testovaní správania sa objektov v scéne si môžeme všimnúť, že loptička pri kolíziách rotuje. To nie je žiadúce. Zakázať túto rotáciu je možné zaškrtnutím *checkboxu* vlastnosti *Freeze Rotation* v komponente *Rigidbody 2D* objektu *Ball*.

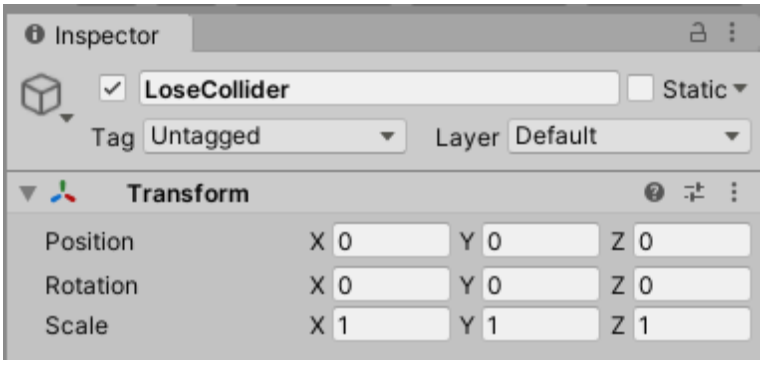


Obr. 86 Zakázanie rotácie objektu *Ball*.

4.4 Zničenie objektu mimo herného priestoru

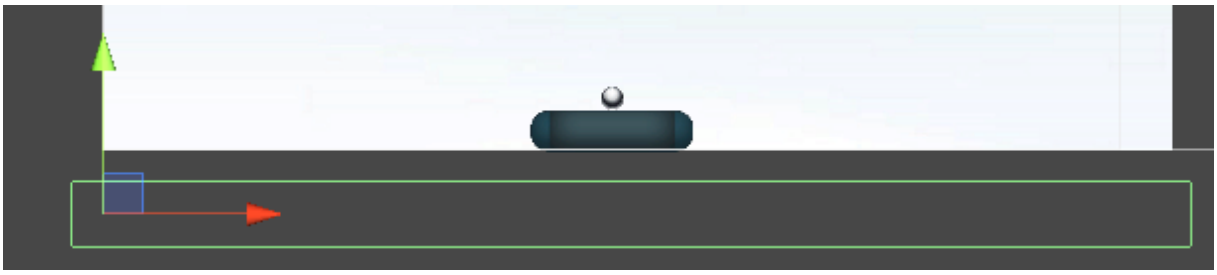
Ďalšia vec, ktorú si môžeme všimnúť pri testovaní hry je, že loptička v prípade ak sa neodrazí od padla alebo iného objektu, začne padať mimo hernú plochu, pričom tento objekt stále existuje, čoho dôkazom je, že objekt *Ball* v okne *Hierarchy* stále figuruje. Tento jav nie je žiadúci. Cieľom je zabezpečiť, aby tento objekt zanikol v prípade ak sa loptička dostane mimo hraciu plochu. Ako prvé si v okne *Hierarchy* vytvoríme prázdny *Game Object*, ktorý pomenujeme *LoseCollider*, resetneme jeho

nastavenia v komponente *Transform* – čím sa pozícia tohto objektu nastaví do bodu s hodnotami $x = 0$ a $y = 0$, t. j. do ľavého spodného rohu.



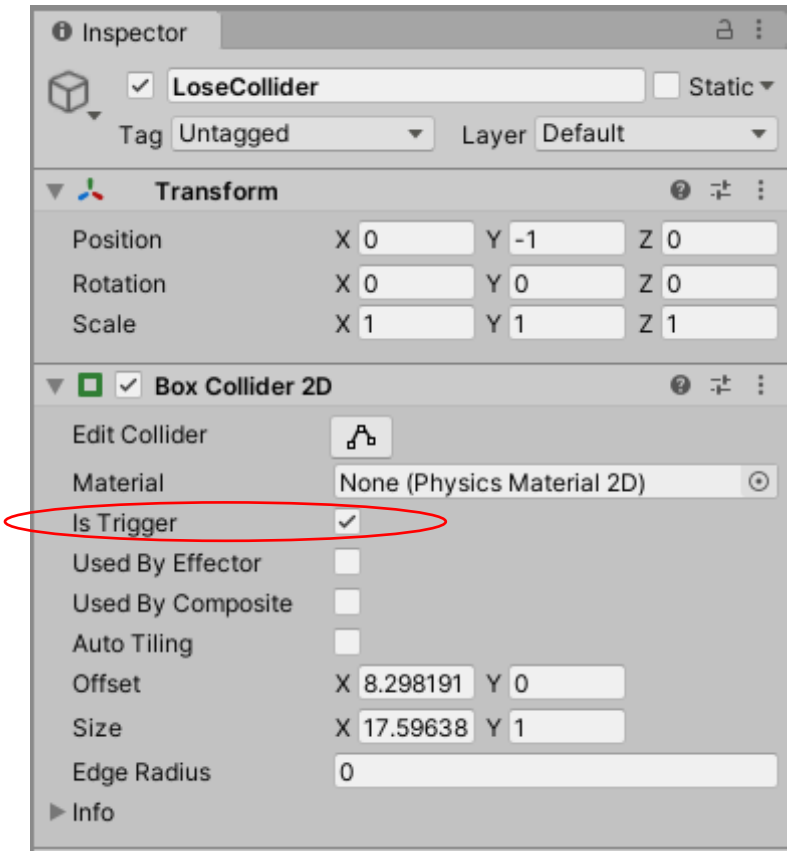
Obr. 87 Vytvorenie prázdneho *Game Objectu* – *LoseCollider*.

Objektu pridáme *Box Collider 2D*, jeho veľkosť pomocou *Edit Collider* nastavíme tak, aby bol o niečo väčší ako herná plocha a objekt umiestnime pod túto plochu, tak ako ilustruje obrázok 88.



Obr. 88 Pridanie *Box Collideru 2D* objektu *LoseCollider*.

V okne *Inspector* pre objekt *LoseCollider* si môžeme všimnúť, že východiskové nastavenie pre vlastnosť *Is Trigger* je neaktívny stav. To znamená, že v prípade stretu dvoch objektov, nastáva medzi nimi kolízia a objekt *Ball* sa od tohto objektu odrazí. Ak nastavíme vlastnosť *Is Trigger* na aktívnu, tak loptička sa neodrazí, ale prejde cez *collider*. Avšak týmto dostaneme možnosť reagovať na tento stav tak ako potrebujeme. *Trigger* nám vlastne umožní pridať požadovanú funkcionalitu tým, že tento stav je detegovaný volaním *OnTriggerEnter*().



Obr. 89 Vlastnosť Is *Trigger*.

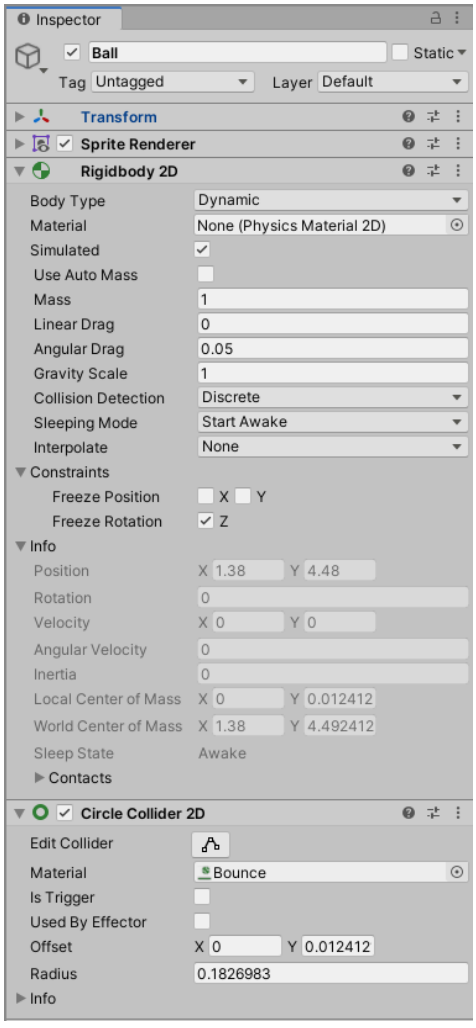
4.4.1 *Collision action matrix*

Pri kolízií dvoch objektov, vzhľadom na rôzne nastavenia týchto objektov môže dôjsť k rôznym udalostiam. Tabuľka uvedená na obrázku 90 špecifikuje, ktoré funkcie na obsluhu udalostí sú vyvolané v závislosti od komponentov, ktoré sú k objektom pridané. Niektoré z kombinácií spôsobia, že kolízia zasiahne iba jeden z dvoch objektov. Avšak, vo všeobecnosti platí, že fyzika sa neaplikuje na objekt, ktorý nemá komponent *Rigidbody*.

Napríklad, ak sa pozrieme na vlastnosti objektu *Ball* (Obr. 91), tak vidíme, že pri použitom *collideri* nemáme zaškrtnutú možnosť *Is Trigger* a *Body Type* v komponente *Rigidbody 2D* je nastavený na *Dynamic*. Z toho vyplýva, že podľa tabuľky je pre nás smerodajná možnosť *Dynamic Rigidbody Collider*. Podľa toho s akým objektom príde do kontaktu, dôjde k vyvolaniu príslušnej udalosti.

	Static Collider	Dynamic Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Dynamic Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider		Collision			Trigger	Trigger
Dynamic Rigidbody Collider	Collision	Collision	Collision	Trigger	Trigger	Trigger
Kinematic Rigidbody Collider		Collision		Trigger	Trigger	Trigger
Static Trigger Collider		Trigger	Trigger		Trigger	Trigger
Dynamic Rigidbody Trigger Collider	Trigger	Trigger	Trigger	Trigger	Trigger	Trigger
Kinematic Rigidbody Trigger Collider	Trigger	Trigger	Trigger	Trigger	Trigger	Trigger

Obr. 90 *Collision action matrix* (Davidson, Tristem. 2019).

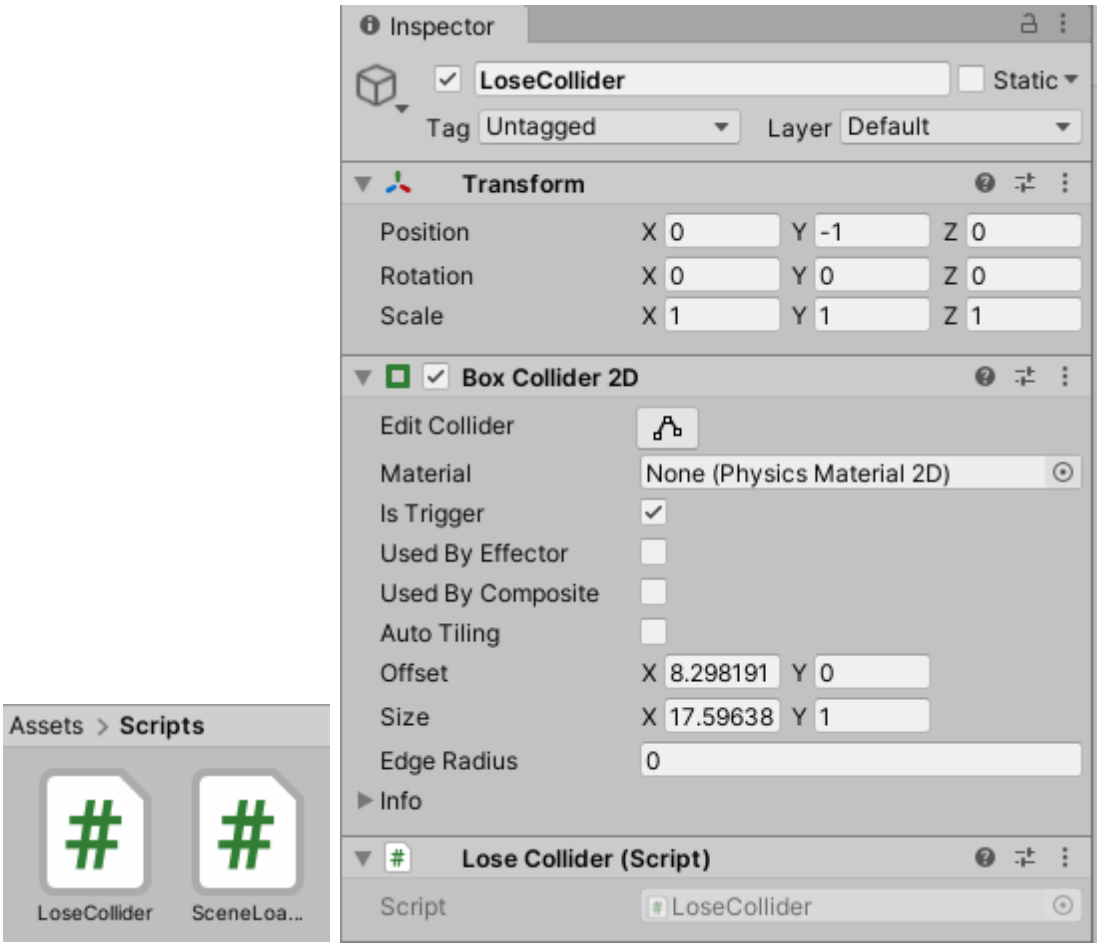


Obr. 91 Okno *Inspector* pre objekt *Ball*.

Pri prvom kontakte objektov sa volá *OnCollisionEnter2D*(). V prípade nekonečnej slučky, t. j. ak hra stále beží, tieto objekty stále kolidujú a vtedy je volaná *OnCollisonStay2D*() a v prípade prerušenia kontaktu sa vyvolá *OnCollisionExit2D*() (Unity, 2020i).

4.4.2 Pridanie interaktivity objektu *LoseCollider*

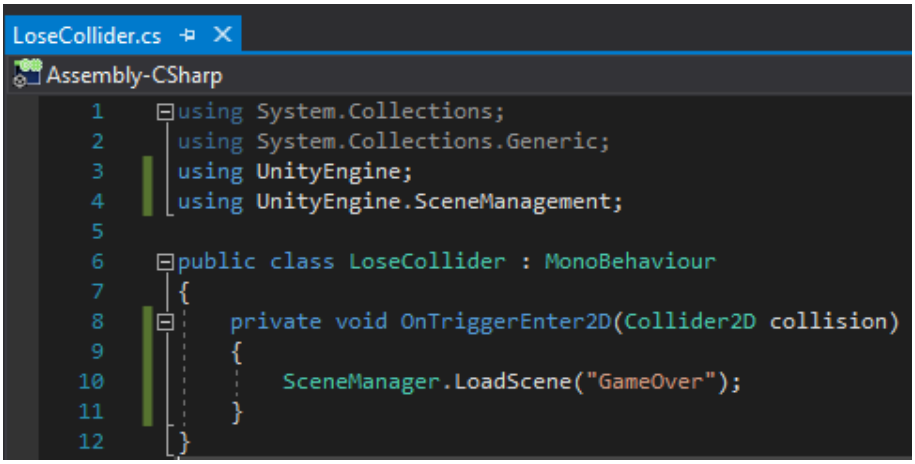
Aby nastalo zničenie objektu *Ball* pri kolízii s objektom *LoseCollider*, potrebujeme túto udalosť obsluhovať pomocou skriptu. Vytvoríme si skript pre objekt *LoseCollider* s rovnakým názvom, ktorý vytvoríme v priečinku *Scripts*, a pridáme ho ako komponent objektu *LoseCollider*.



Obr. 92 Vytvorenie C# skriptu a pridanie tohto komponentu objektu *LoseCollider*.

V skripte implementujeme *OnTriggerEnter2D*(), ktorá zabezpečí načítanie scény s názvom *GameOver*. Jedným z pravidiel hry je, že ak sa nepodarí loptičku odraziť

padlom, dochádza k ukončeniu hry. Pre načítanie scény, obdobne ako v kapitole 2.2.1 použijeme metódu *LoadScene*(). V tomto prípade však využijeme prototyp, kde scéna je určená jej názvom (*string reference*) a nie indexom. V skripte nesmieme zabudnúť na pridanie menného priestoru *SceneManagement*. Funkciu *Start*() a *Update*() v tomto skripte nepotrebujeme (Obr. 93).



Obr. 93 Vizuál skriptu *LoseCollider.cs*.

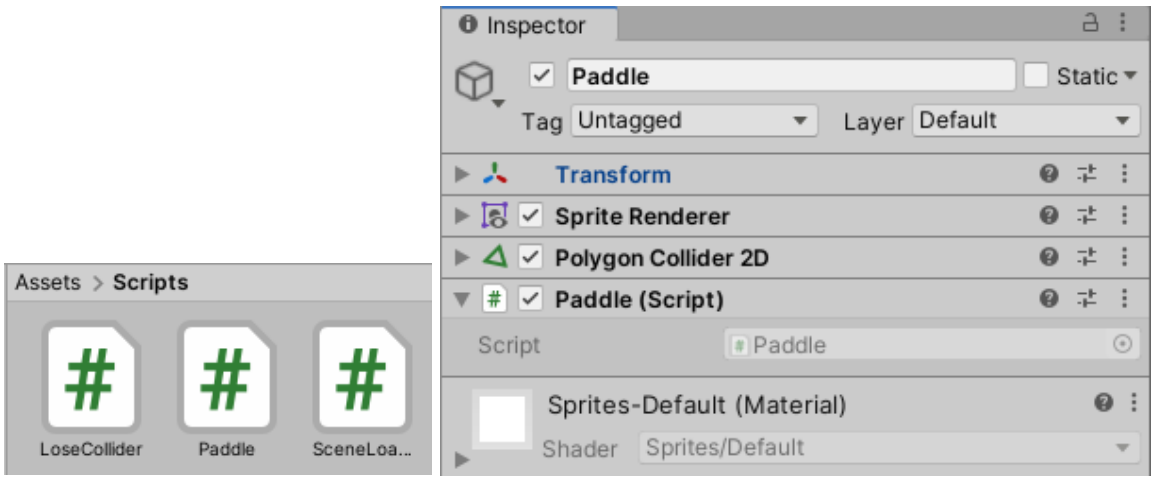
4.5 Kontrolné otázky a úlohy

1. Čo vyjadruje termín *collision*?
2. Z akého dôvodu pridávame objektom komponent *Rigidbody 2D*?
3. Podľa vlastnosti *Body Type*, aké možnosti nastavenia *Rigidbody* poznáme?
4. Čo vyjadruje termín *collider*? Aké typy *colliderov* poznáte?
5. Z akého dôvodu pridávame objektom komponent *Physics Material 2D*?
6. Aká je funkcionalita vlastnosti *Is Trigger*?
7. Ako sa volajú funkcie na obsluhu udalostí, ktoré sú volané v prípade kolízie dvoch objektov?

5 Zabezpečenie pohybu objektu *Paddle* pomocou myši

Našim cieľom je zabezpečiť ovládanie padla v smere osi *x* s pomocou pohybu myši. Aby sme túto funkcionalitu zabezpečili je dôležité identifikovať pozíciu myši pri pohybe, konvertovať túto pozíciu do jednotiek sveta (*world units*) a modifikovať pozíciu pri každom pohybe myši.

Zmenu pozície padla je nutné modifikovať dynamicky, preto si vytvoríme nový skript s názvom *Paddle* a pridáme ho objektu padla (Obr. 94).

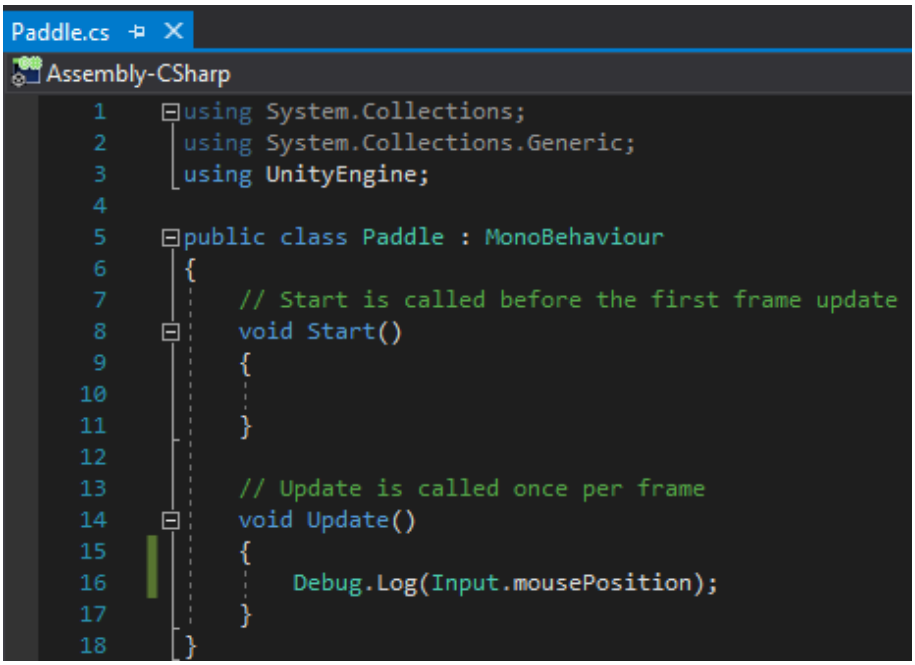


Obr. 94 Vytvorenie C# skriptu a pridanie tohto komponentu objektu *Paddle*.

Na oboznámenie sa s tým, ako sa menia súradnice (koordináty) pozície pri pohybe myšou vo svete Unity je možné využiť metódu *Debug.Log()*. S jej pomocou vieme vypisovať požadovaný obsah do konzoly (*Unity Console*). Táto metóda sa často používa pre ladenie (*debugovanie*) akýchkoľvek stavov počas vývoja. Zavolaním tejto metódy vo funkcii *Update()* v tvare:

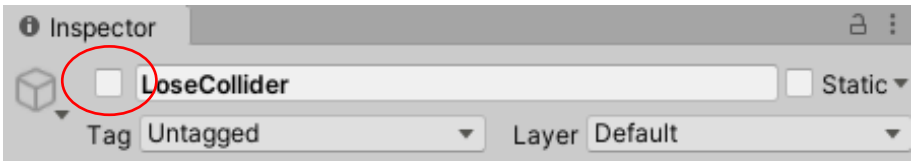
```
Debug.Log(Input.mousePosition);
```

môžeme sledovať, ako sa jednotlivé hodnoty pri pohybe myšou menia v konzole pri spustení hry (Obr. 97).

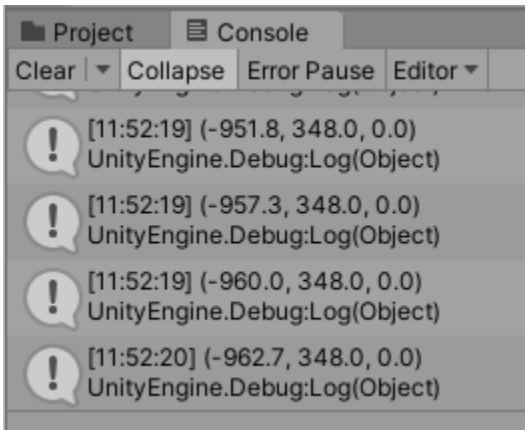


Obr. 95 Vizuál skriptu *Paddle.cs*.

V kapitole 4.4.2 sme objektu *LoseCollider* pridali funkcionalitu zobrazenia scény *GameOver* v prípade, ak nedôjde k odrazeniu loptičky padlom. Aby sme mohli sledovať ako sa menia pozície myši (Obr. 97) bez toho aby nastalo ukončenie hry, stačí deaktivovať objekt *LoseCollider* jednoduchým odškrtnutím *checkboxu* (Obr. 96).



Obr. 96 Deaktivácia objektu *LoseCollider*.



Obr. 97 Výpis súradníc pri pohybe myšou v konzole.

V prípade ak chceme sledovať zmenu len v smere osi *x*, stačí upraviť parameter metódy takto:

```
Debug.Log(Input.mousePosition.x);
```

Pri testovaní si môžeme všimnúť, že hodnoty sa pohybujú v rozmedzí cca 0 ÷ −1345, čo nie je presne hodnota, ktorú by väčšina čitateľov očakávala. Ak však hodnotu predelíme šírkou obrazovky, tak sa dostaneme na hodnoty v rozsahu cca 0 ÷ 1, čo už má väčšiu logiku, pretože hodnota v strede obrazovky predstavuje hodnotu 0,5.

```
Debug.Log(Input.mousePosition.x/Screen.width);
```

Ak tento výraz vynásobíme hodnotou 16 (herný svet predstavuje 16 *units* horizontálne vzhľadom na veľkosť kamery a použitý *Aspect Ratio*), budeme pracovať v rozsahu 0 ÷ 16. Použitie takéhoto rozsahu nám príde logickejšie. Stred obrazovky teda reprezentuje hodnota 8.

```
Debug.Log(Input.mousePosition.x/Screen.width*16);
```

Pri žiadnom programovaní nie je vhodné aby sme pracovali priamo s číselnými hodnotami namiesto premenných – znižujeme tým modifikovateľnosť riešenia. Je preto vhodné aby sme zaviedli premennú, ktorá bude predstavovať počet *units* reprezentujúcich šírku obrazovky – *screenWidthInUnits*, ktorú v našom prípade inicializujeme hodnotou 16 (Obr. 98).

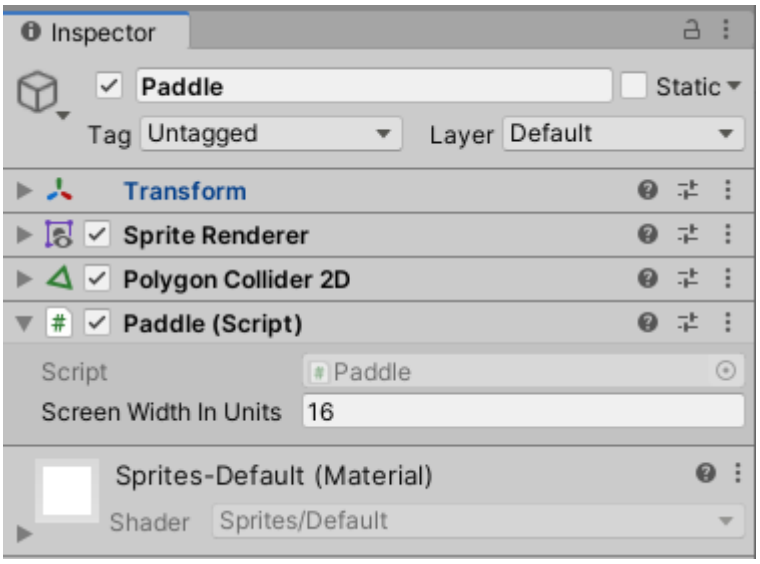
Paddle.cs

Assembly-CSharp

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Paddle : MonoBehaviour
6   {
7       [SerializeField] float screenWidthInUnits = 16f;
8       // Start is called before the first frame update
9       void Start()
10      {
11
12      }
13
14      // Update is called once per frame
15      void Update()
16      {
17          Debug.Log(Input.mousePosition.x/Screen.width*screenWidthInUnits);
18      }
19  }
```

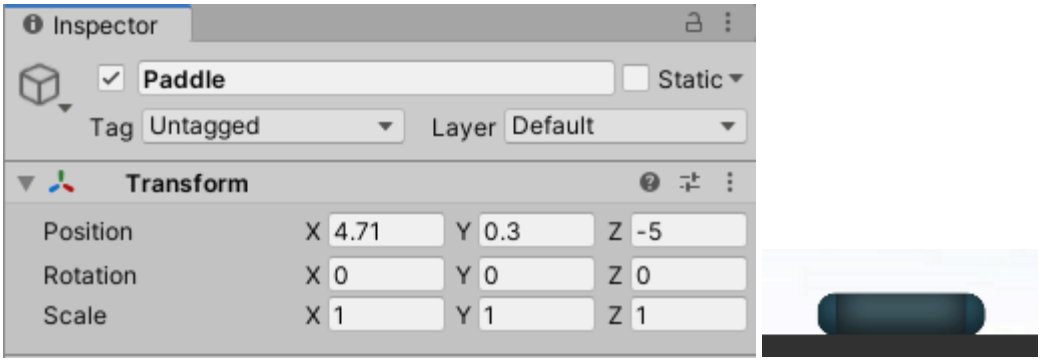
Obr. 98 Vizuál skriptu *Paddle.cs*.

Pozn. autora: Atribút *SerializeField* používame pri premenných, ktoré sú *private*, avšak chceme aby sa táto premenná zobrazila v editore Unity s cieľom jej jednoduchej modifikácie pri testovaní (Obr. 99). V prípade zmeny rozlíšenia, alebo modifikácie pre *Aspect Ratio*, by bolo nutné túto hodnotu modifikovať.



Obr. 99 Premenná *screenWidthInUnits* zobrazená v editore Unity.

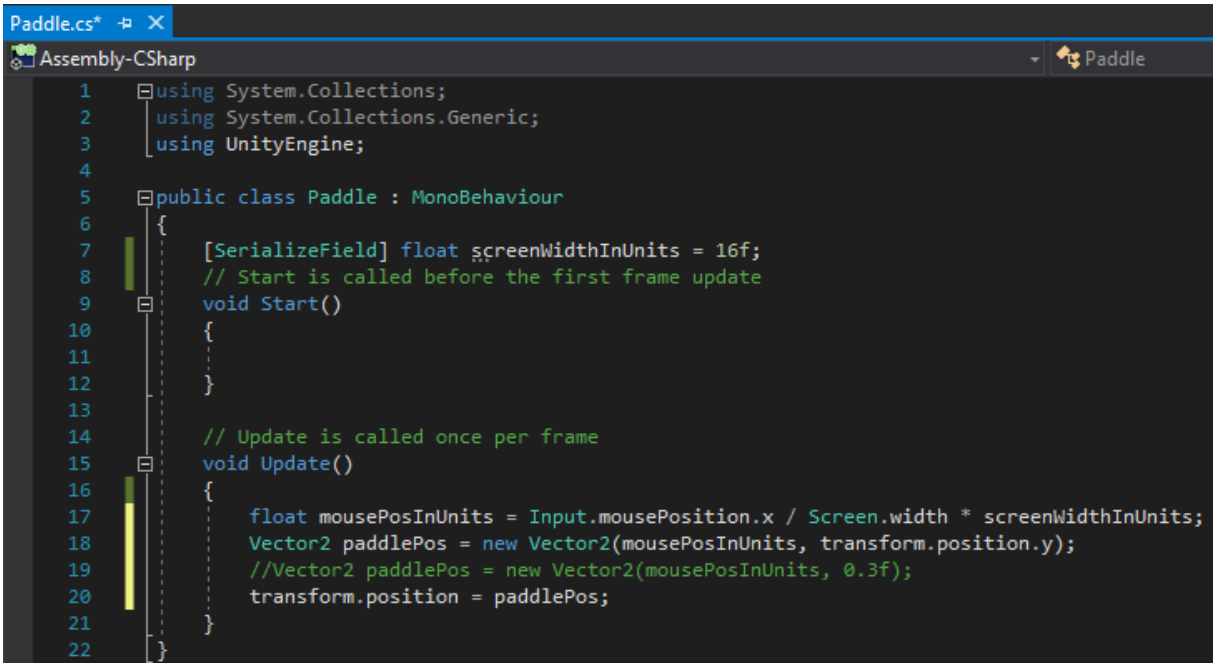
Následne upravíme pozíciu objektu *Paddle* tak, aby padlo bolo na zemi, v našom prípade $y = 0{,}3$.



Obr. 100 Úprava pozície padla v smere osi y.

Je zrejmé, že poloha objektu *Paddle* je reprezentovaná súradnicami x a y, pričom hodnota x je závislá od pohybu myšou a hodnota y je nemenná. V Unity sa na reprezentáciu takéhoto typu premennej používa štruktúra Vector2. Vo funkcii *Update*() preto definujeme premennú typu Vector2 s názvom *paddlePos*. Zmenu súradníc robíme cez vlastnosť *transform.position*.

Premenná *mousePosInUnits* reprezentuje prepočítanú hodnotu *x*–ovej súradnice myši do jednotiek sveta (*units*).

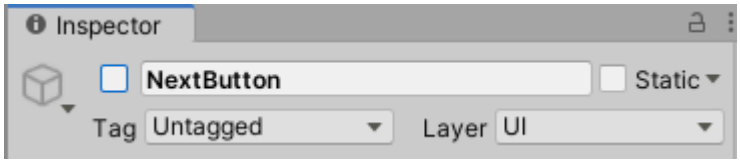


Obr. 101 Vizuál skriptu *Paddle.cs*.

Pozn. autora: Hodnotu v smere osi *y* je možné nastaviť buď priamo, ako je vidieť v zakomentovanom príkaze na obrázku 101 riadok 19, alebo pomocou vlastnosti – *transform.position.y*. Odporúčame používať druhý spôsob, ktorý bude reagovať na zmeny, ktoré by ste v prípade potreby spravili v editore Unity.

Pri testovaní hry sa vám môže stať, že padlo nebudeme vidieť, t. j. je prekryté pozadím. Je to spôsobené tým, že hodnota v smere osi *z* objektu *Paddle* sa zmení v tomto prípade na 0 (východisková hodnota), rovnako ako má aj použité pozadie. Pre vyriešenie stačí pozadie posunúť v smere osi *z*, napr. na hodnotu 5.

Ak vám pri testovaní vadí zobrazenie tlačidla *NextLevel*, tak ho môžete deaktivovať rovnakým spôsobom ako sme vyššie urobili pri objekte *LoseCollider*.



Obr. 102 Deaktivácia objektu *NextButton*.

5.1 Ohraničenie pohybu padla

Ako sme testovaním zistili, padlo je momentálne schopné sa dostať aj mimo obrazovky, či už vľavo alebo vpravo. K dispozícii máme metódu *Mathf.Clamp()*, ktorá vráti hodnotu typu *float* v zadanom intervale. Ak prekročí stanovenú minimálnu hodnotu, vráti hodnotu tohto minima a v prípade ak prekročí maximálnu stanovenú hodnotu vráti hodnotu maxima. Využitím tejto metódy zabezpečíme obmedzenie pohybu padla. Premenné *min*X a *max*X určujúce hranice intervalu nastavíme ako *SerializeField* s cieľom ich jednoduchej modifikácie priamo v editore Unity (Obr. 103).

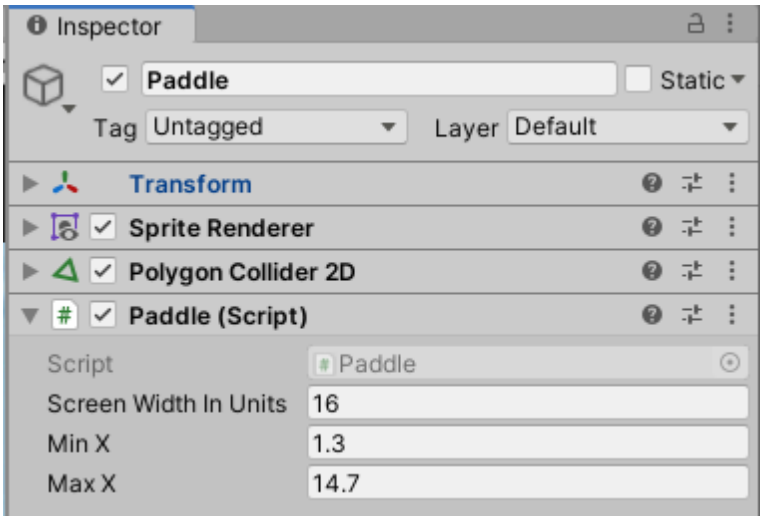
Paddle.cs*

Assembly-CSharp — Paddle

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Paddle : MonoBehaviour
6   {
7       //configuration parameters
8       [SerializeField] float screenWidthInUnits = 16f;
9       [SerializeField] float minX = 1f;
10      [SerializeField] float maxX = 15f;
11      // Start is called before the first frame update
12      void Start()
13      {
14
15      }
16
17      // Update is called once per frame
18      void Update()
19      {
20          float mousePosInUnits = Input.mousePosition.x / Screen.width * screenWidthInUnits;
21          mousePosInUnits = Mathf.Clamp(mousePosInUnits, minX, maxX);
22          Vector2 paddlePos = new Vector2(mousePosInUnits, transform.position.y);
23          transform.position = paddlePos;
24      }
25  }
```

Obr. 103 Vizuál skriptu *Paddle.cs*.

Pri testovaní funkčnosti pravdepodobne zistíte, že inicializácia hodnôt *min*X = 1f a *max*X = 15f nie je ideálna. Počas testovania je možné tieto hodnoty upravovať priamo v editore Unity a testovať hodnoty, ktoré sú vyhovujúce. Pozor však na to, že tieto zmeny sa neprejavia (neuložia) a po zastavení testovania ich musíte nastaviť. My sme museli vzhľadom na veľkosť padla hodnoty upraviť takto:



Obr. 104 Modifikácia premenných minX a maxX.

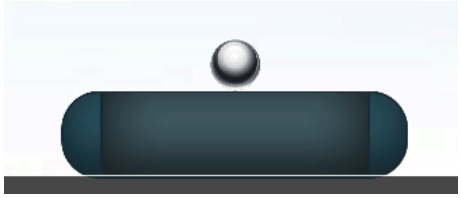
5.2 Kontrolné otázky a úlohy

1. Pomocou akej metódy vieme vypisovať informácie do konzoly?
2. Definujte dátový typ *Vector2*.
3. Akým príkazom je možné zrealizovať zmenu pozície objektu?
4. Definujte metódu *Mathf.Clamp*().
5. Čo znamená ak premenné deklarujeme ako *SerializeField*?

6 Zabezpečenie správania sa objektu *Ball*

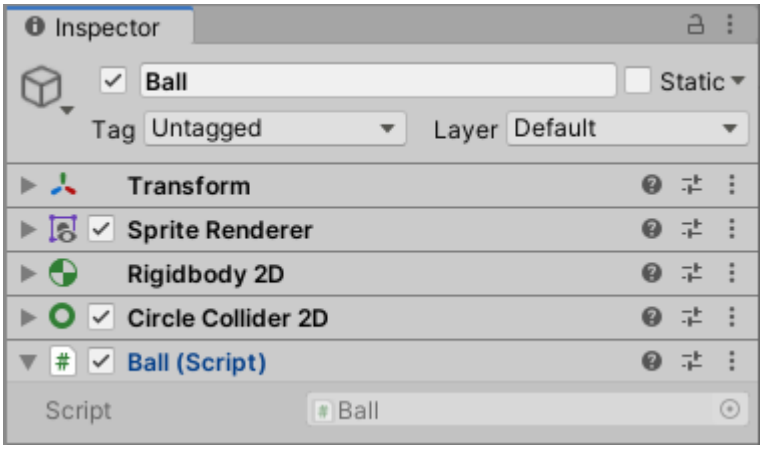
V tejto časti zabezpečíme, aby na začiatku hry loptička sedela na padle aj keď budeme pohybovať padlom pomocou myši a po určitom čase, resp. akcii ju vystrelíme.

Ako prvé umiestnime loptičku na padlo zmenou jej pozície v smere osi x a y priamo v editore Unity (Obr. 105).



Obr. 105 Zmena pozície objektu *Ball*.

Následne vytvoríme skript s názvom *Ball* a pridáme ho objektu *Ball* (Obr. 106).

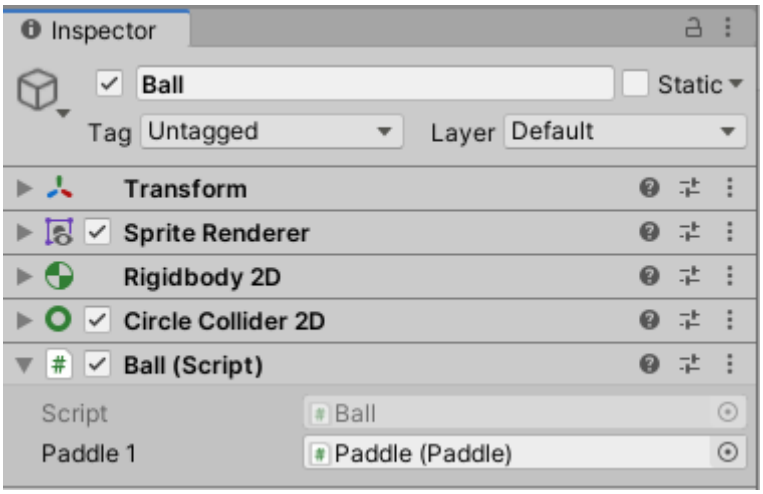


Obr. 106 Pridanie skriptu *Ball.cs* objektu *Ball*.

V skripte vytvoríme premennú, ktorá bude identifikovať padlo. Preto je táto premenná typu *Paddle*. Cieľom je zabezpečiť interakciu (súčasný pohyb) medzi objektmi *Ball* a *Paddle*.

```
[SerializeField] Paddle paddle1;
```

Ako identifikátor pre túto premennú sme zvolili *paddle1*, pretože pri rozšíreniach hry je možné počítať aj s tým, že hra bude obsahovať viacero objektov typu *Paddle*. Po doplnení kódu sa vrátime do editora Unity a pridáme objekt *Paddle* k práve vytvorenej premennej (Obr. 107).



Obr. 107 Inicializácia premennej *paddle1* v editore Unity.

Pre zabezpečenie interakcie medzi loptičkou a padlom je nutné vypočítať vzdialenosť medzi týmito dvomi objektmi. Zistiť pozíciu objektu *Ball* je možné priamo pomocou:

```
transform.position
```

Vzhľadom na to, že pracujeme v skripte pre objekt *Ball*, nie je nutná žiadna detailnejšia špecifikácia, na rozdiel od zisťovania pozície pre objekt *Paddle*.

Vzdialenosť objektov vypočítame rozdielom pozícii týchto objektov, ktorý uložíme do premennej typu *Vector2*:

```
paddleToBallVector = transform.position - paddle1.transform.position;
```

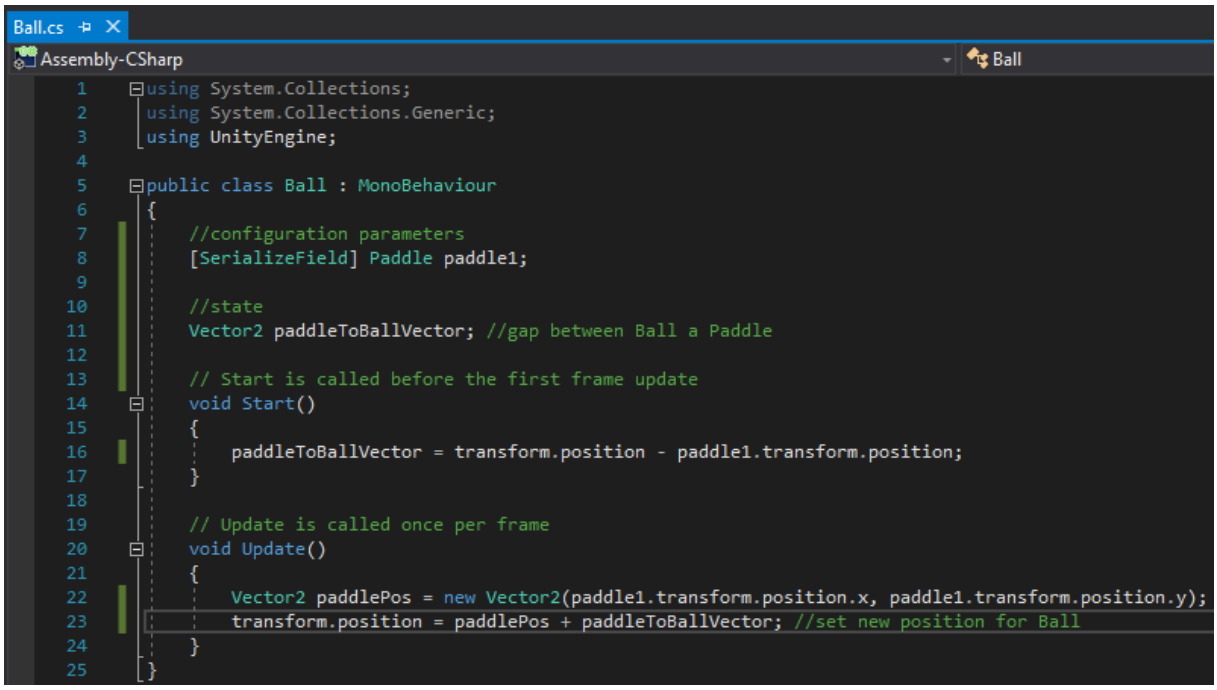
Vo funkcii *Update()* definujeme ďalšiu premennú typu *Vector2* pre pozíciu padla:

```
Vector2 paddlePos =
        new Vector2(paddle1.transform.position.x, paddle1.transform.position.y);
```

Pozíciu loptičky upravíme tak, že bude priamo závislá od pozície padla, ku ktorej pripočítame rozdiel uložený v premennej *paddleToBallVector*:

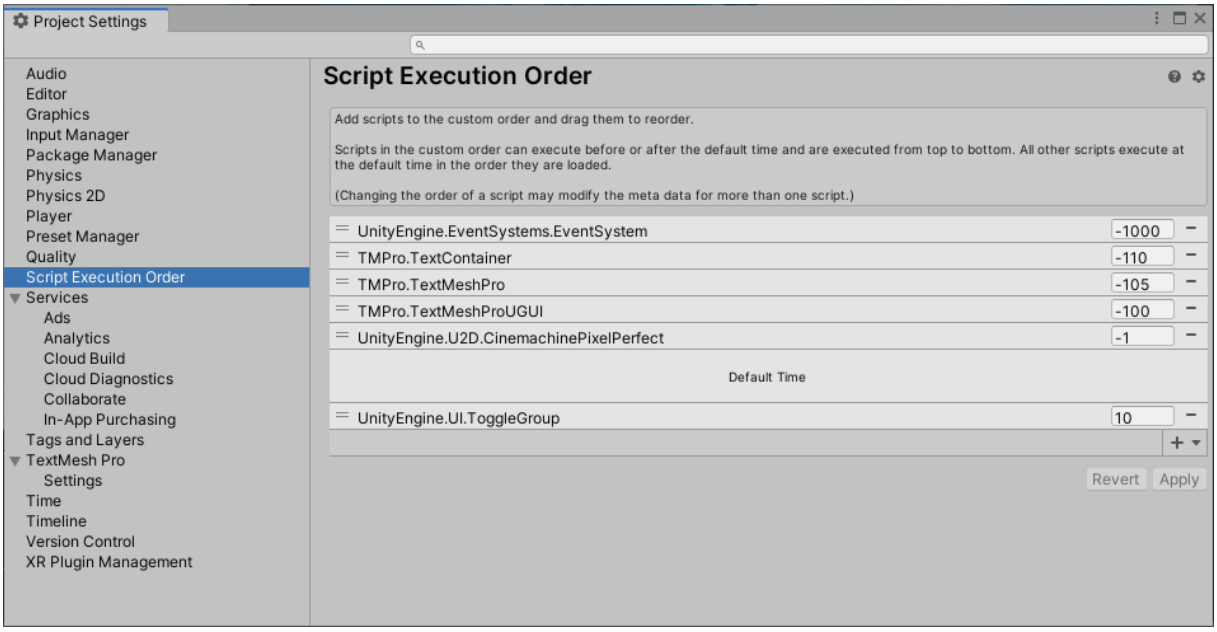
```
transform.position = paddlePos + paddleToBallVector;
```

Skript *Ball.cs* ilustruje obrázok 108.



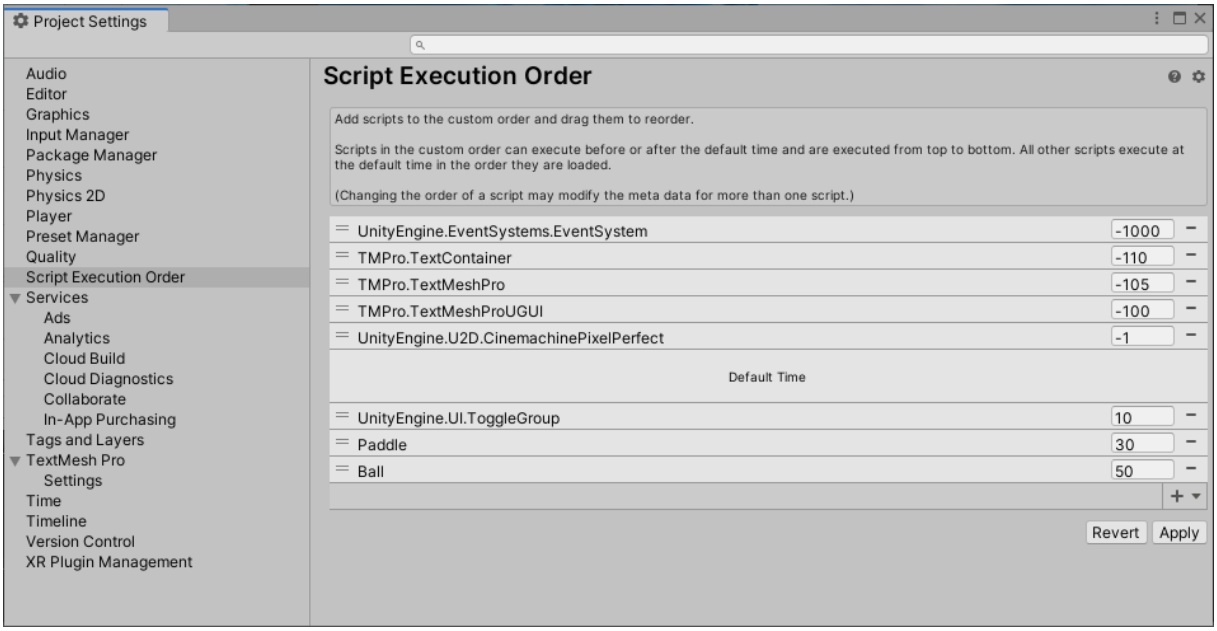
Obr. 108 Vizuál skriptu *Ball.cs*.

Pozn. autora: Ak sa vám pri testovaní hry zdá, že pri veľmi rýchlom pohybe padla sa loptička ako keby kĺzala po padle, môže to byť spôsobené tým, že výpočty pre padlo a loptičku sa realizujú v rôznom čase. Toto je možné upraviť pomocou *Edit → Project Settings*, konkrétne v nastaveniach kategórie *Script Execution Order* (Obr. 109).



Obr. 109 *Script Execution Order*.

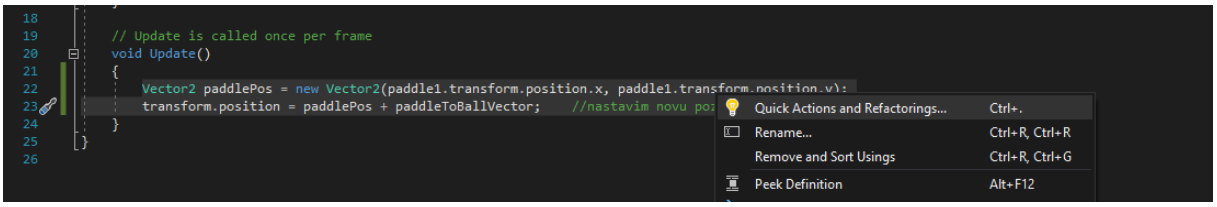
Pomocou tlačidla „+“ je možné pridať skripty pre objekty *Paddle* a *Ball*. Hodnoty uvedené pri týchto objektoch nemajú nejaký významný charakter – ide o relatívne hodnoty. Dôležité je aby skript *Paddle* mal nižšiu hodnotu, ako skript *Ball*, čím zabezpečíme, že ako prvé sa budú realizovať výpočty pre objekt *Paddle* a následne pre objekt *Ball*. Unity realizuje vykonávanie podľa tohto zoznamu od hora dole, resp. podľa uvedených hodnôt od tých ktoré majú najvyššie negatívne hodnoty, smerom k tým, ktoré majú najvyššie kladné hodnoty (Unity, 2020l). Zmeny treba aplikovať pomocou *Apply*.



Obr. 110 Zmena v *Script Execution Order*.

6.1 Vystrelenie objektu *Ball*

Vystrelenie loptičky podmienime stlačeniu ľavého tlačidla myši. Vytvoríme dve metódy. Jednu, ktorá prichytí loptičku na padle, čo sme v podstate spravili krok pred týmto a druhú, ktorá loptičku vystrelí. Keďže jadro prvej metódy máme nachystané, najjednoduchší spôsob vytvorenia metódy je, že si označíme daný kód a využijeme možnosť kontextovej ponuky *Quick Actions and Refactoring* vyvolanej stlačením pravého tlačila myši:



Obr. 111 Vytvorenie metódy z existujúceho kódu – krok 1.

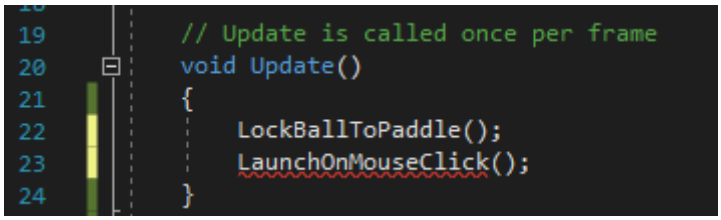
Potom stlačíme *ExctractMethod*:

Obr. 112 Vytvorenie metódy z existujúceho kódu – krok 2.

A určíme názov metódy *LockBallToPaddle()*:

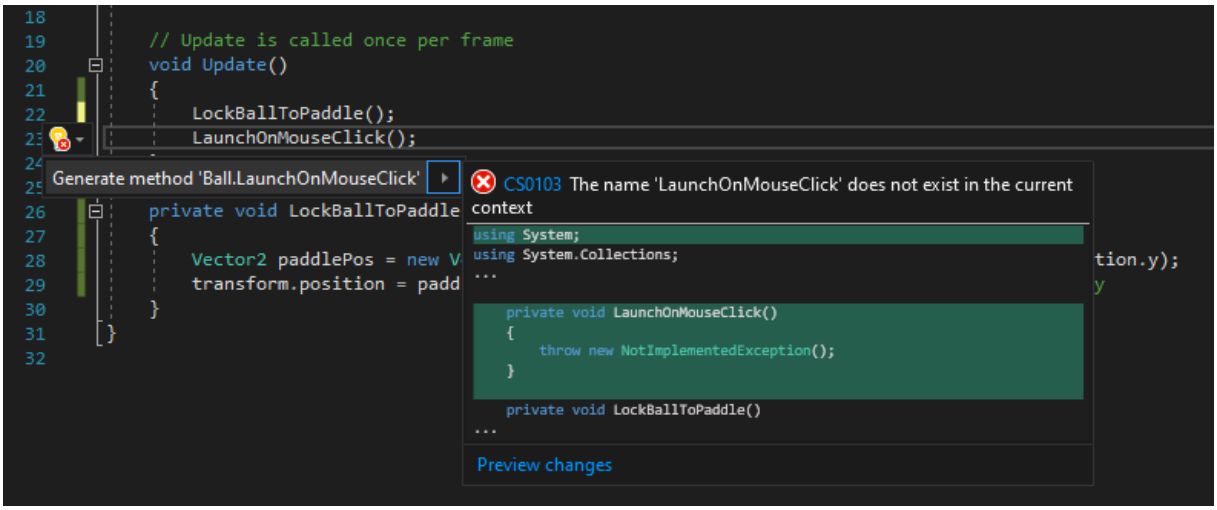
Obr. 113 Vytvorenie metódy z existujúceho kódu – krok 3.

Funkcia *Update()* bude okrem volania tejto metódy obsahovať aj volanie novovytvorenej metódy *LaunchOnMouseClick()*, ktorú si následne vytvoríme:



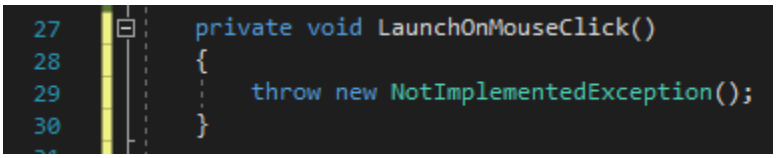
Obr. 114 Zavolanie zatiaľ nedefinovanej metódy.

Opätovne si vieme kódovanie urýchliť vygenerovaním metódy pomocou kontextovej ponuky *Quick Actions and Refactoring – Generate method* (Obr. 115). Túto metódu si môžeme napísať aj priamo.



Obr. 115 Vygenerovanie metódy pomocou kontextovej ponuky.

Ošetrenie výnimky (Obr. 116 riadok 29) môžeme vymazať, to momentálne nepotrebujeme.



Obr. 116 Výsledok po vygenerovaní metódy.

Skôr ako začneme implementovať telo tejto metódy, je dôležité zistiť, akú metódu triedy *Input* použiť, aby sme mohli podmieniť vystrelenie loptičky stlačeniu tlačidla myši. K dispozícii máme metódu *GetMouseButtonDown()*, ktorá vráti logickú hodnotu pravdy v prípade ak nastalo stlačenie tlačidla myši. Metóda má práve jeden parameter, ktorý určuje ktoré tlačidlo myši bolo stlačené. V prípade použitia ľavého tlačidla myši je tento parameter rovný 0.

```
Input.GetMouseButtonDown(0)
```

Vystrelenie loptičky zabezpečíme zmenou hodnôt vlastnosti *velocity* komponentu *Rigidbody 2D*. Keďže nie každý objekt musí obsahovať tento komponent, je prístup k nastaveniam mierne odlišný ako sme používali napr. v prípade nastavenia pozície objektu pomocou *transform.position*. Komponent *Transform* totiž obsahuje každý

objekt. K prístupu použijeme metódu *GetComponent()* triedy *GameObject* pričom v < > špecifikujeme ku ktorému komponentu chceme pristúpiť, v tomto prípade *Rigidbody 2D*. Následne určíme vlastnosť, ktorú chceme modifikovať, t. j. *velocity*. Tá sa nastavuje pomocou vektora, preto priradíme vektor s konkrétnymi hodnotami, pričom hodnoty uložíme do nových premenných *xPush* a *yPush*. Po logickej úvahe, že hranice sveta sa pohybujú v rozmedzí 0 ÷ 16, tieto premenné inicializujeme hodnotou 2 pre *xPush* a hodnotou 15 pre *yPush*. Tieto premenné nastavíme ako *SerializeField*, pretože pri testovaní funkcionality budeme s týmito hodnotami experimentovať tak, aby sme našli ich najvhodnejšie nastavenia:

```
[SerializeField] float xPush = 2f;
[SerializeField] float yPush = 15f;

GetComponent<Rigidbody2D>().velocity = new Vector2(2f, 15f);
```

Celá metóda má tvar:

```
private void LaunchOnMouseClick()
{
    if (Input.GetMouseButtonDown(0))
    {
        GetComponent<Rigidbody2D>().velocity = new Vector2(xPush, yPush);
    }
}
```

Ak otestujeme funkčnosť kódu v Unity, k žiadnemu vystreleniu loptičky po stlačení ľavého tlačidla myši nedôjde. Prečo? Pretože vo funkcii *Update()* máme povedané, že každý *frame* sa má loptička uzamknúť na padlo – volanie metódy *LockBallToPaddle()*. Ak však chceme, aby loptička vystrelila, toto sa logicky nemôže vykonať.

Cieľom je teda zabezpečiť správanie tak, že kým nebude stlačené ľavé tlačidlo myši, bode loptička prichytená na padle a po stlačení tlačidla sa vystrelí. Toto zabezpečíme zavedením logickej premennej *hasStarted*, ktorá bude inicializovaná na hodnotu *false*, pričom jej hodnota sa zmení po stlačení tlačidla myši. Takto vieme jednoducho sledovať jednotlivé stavy a pomocou podmienky na ne reagovať.

Definujeme novú premennú:

```
bool hasStarted = false;
```

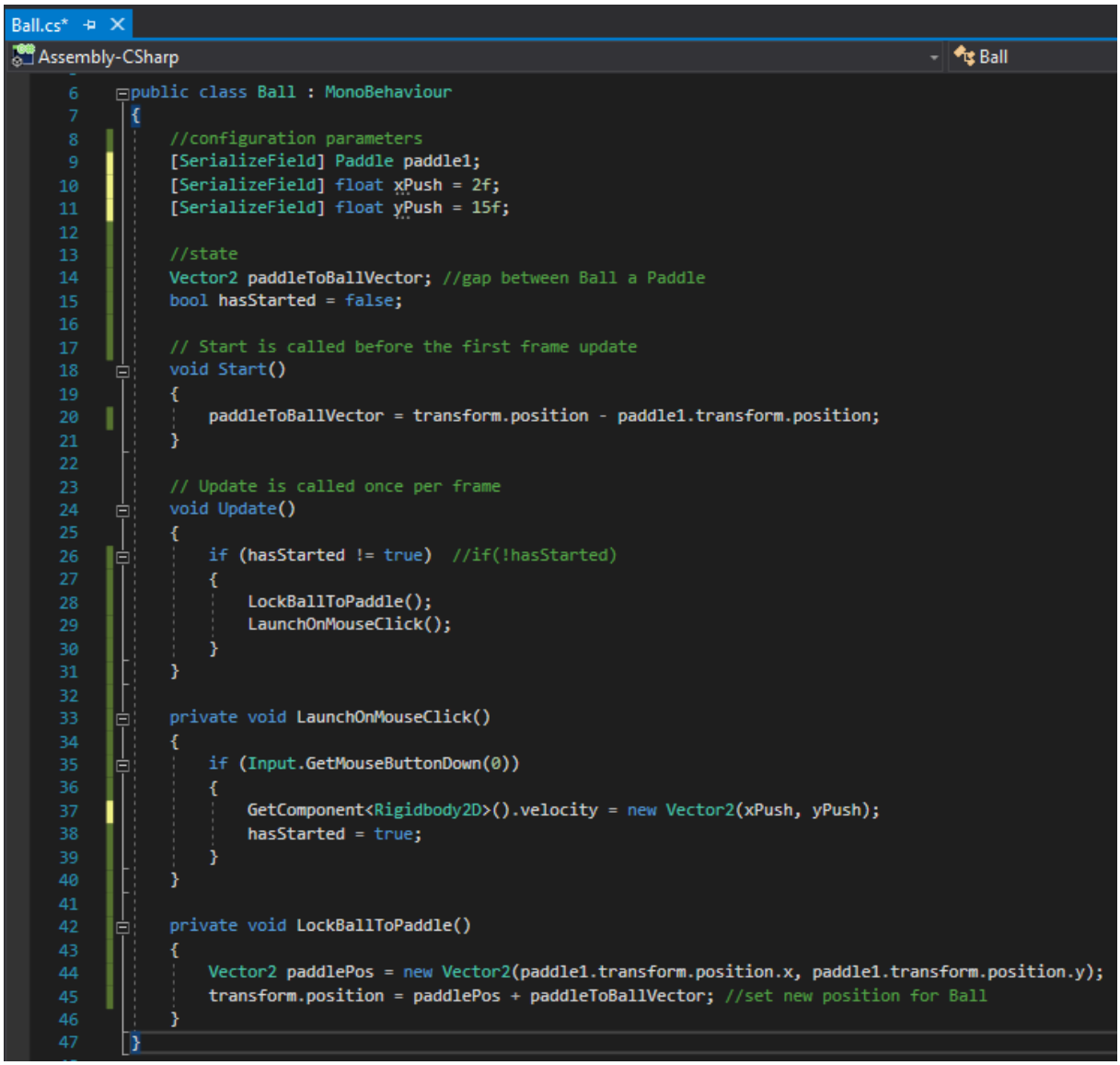
Metódu *LaunchOnMouseClick()* rozšírime o zmenu stavu tejto premennej, t. j. aby sme vedeli detegovať tento stav v prípade ak sa hráč rozhodne loptičku vystreliť:

```
private void LaunchOnMouseClick()
{
    if (Input.GetMouseButtonDown(0))
    {
        GetComponent<Rigidbody2D>().velocity = new Vector2(xPush, yPush);
        hasStarted = true;
    }
}
```

A ako posledný krok upravíme telo funkcie *Update()*, kde podmienime volanie metód hodnote premennej *hasStarted*, t. j. v prípade ak ešte neprišlo k vystreleniu loptičky, táto má byť prichytená k padlu.

```
void Update()
{
    if (hasStarted != true)  //if(!hasStarted)
    {
        LockBallToPaddle();
        LaunchOnMouseClick();
    }
}
```

Kompletná implementácia v skripte *Ball.cs* je ilustrovaná obrázkom 117:

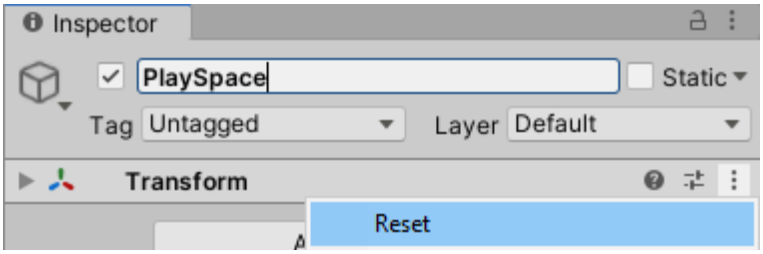


Obr. 117Vizuál skriptu *Ball.cs*.

Všetky zmeny uložíme a testujeme funkčnosť v Unity. Loptičku by sme mali vedieť vystreliť a mala by sa odrážať od padla. Ak nám spadne, malo by dôjsť k načítaniu záverečnej scény – teda ak sme aktivovali objekt *LoseCollider*, ktorý sme pre testovacie účely pred tým deaktivovali. Zmenou hodnôt premenných *xPush* a *yPush* zistíme, že pri zväčšovaní ich hodnôt loptička vystrelí príliš rýchlo a dostane sa mimo herný svet, a pri ich zmenšovaní je zase odraz loptičky pod hranicu herného sveta. Inicializovanie hodnôt premenných, *xPush* = 2 a *yPush* = 15 sa javí ako optimálne.

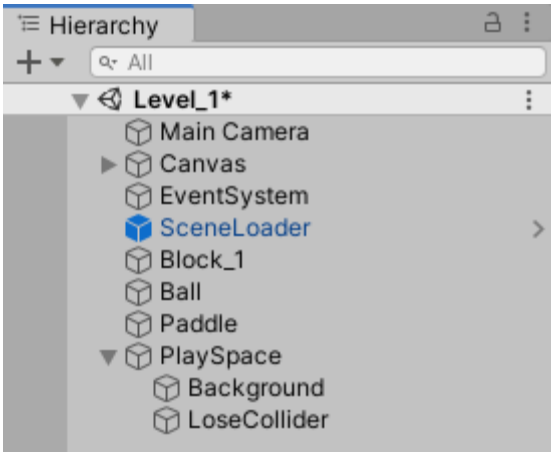
6.2 Ohraničenie herného sveta

Pri testovaní ste si určite všimli, že loptička je schopná dostať sa mimo herný svet. Je to prirodzené, keďže zatiaľ sme zabezpečili len zničenie objektu *Ball* pomocou objektu *LoseCollider*, v prípade ak sa nepodarí loptičku odraziť padlom. Našou úlohou je vytvoriť steny, ktoré ohraničia herný priestor. V okne *Hierarchy* vytvoríme prázdny *Game Object*, pomenujeme ho *PlaySpace* a resetneme nastavenia v *Transform*.



Obr. 118 Vytvorenie objektu *PlaySpace* a resetnutie jeho pozície a rotácie.

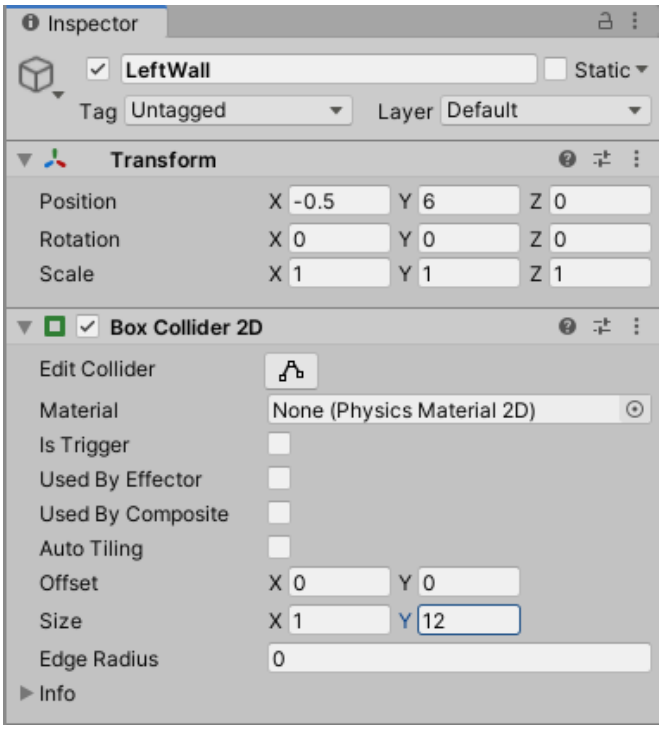
Objekty *Background* a *LoseCollider* zmeníme na potomkov tohto objektu. Zrealizujeme to jednoduchým presunutím objektov v okne *Hierarchy* (Obr. 119).



Obr. 119 Vytvorenie potomkov objektu *PlaySpace*.

Ako potomka objektu *PlaySpace* vytvoríme aj nový *Game Object* a pomenujeme ho *LeftWall*. Pridáme mu *Box Collider 2D*, ktorého veľkosť v smere osi *y* nastavíme na hodnotu 12, keďže náš svet vo vertikálnom smere predstavuje 12 *units*. Pozíciu v smere osi *y* zmeníme na 6 a v smere osi *x* na –0,5 tak aby sme *collider* nastavili ako našu ľavú stenu (Obr. 120).

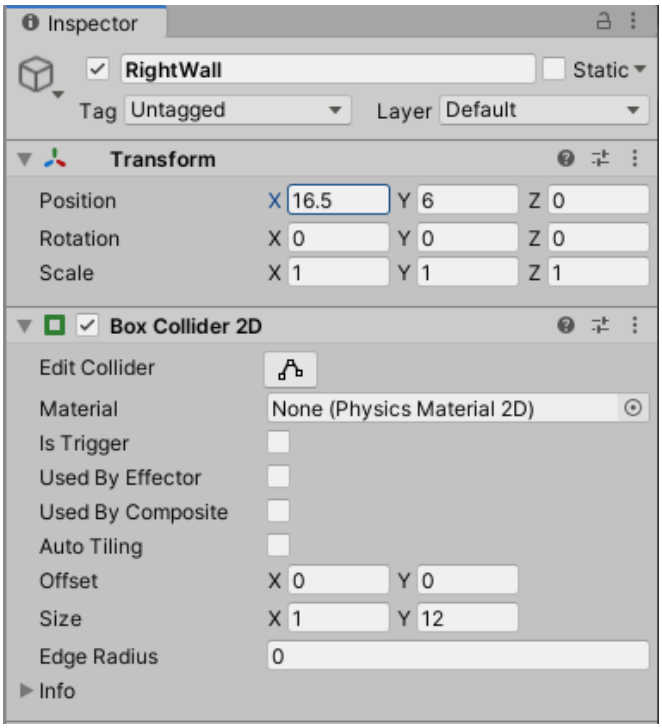
Pozn. autora: Tieto nastavenia sú závislé od toho, že pivot objektu je v strede. V prípade zmeny pozície pivota, by nastavenia pozícií boli iné.



Obr. 120 Nastavenia objektu *LeftWall*.

Obr. 121 Vizuál objektu *LeftWall* v scéne.

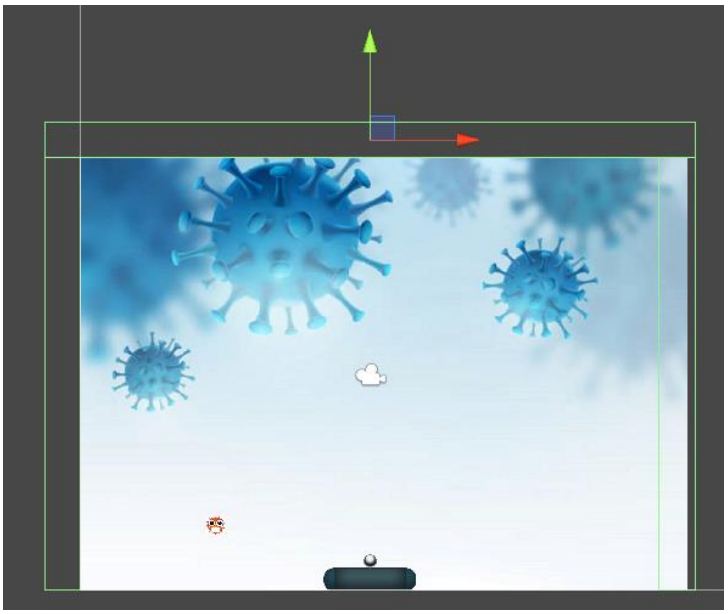
Obdobným spôsobom vytvoríme aj pravú stenu. Najjednoduchší spôsob vytvorenia tohto objektu je duplikovaním Ctrl + D už existujúceho objektu. Nastavenia objektu *RightWall* ilustruje obrázok 122 a jeho vizuál obrázok 123.



Obr. 122 Nastavenia objektu *RightWall*.

Obr. 123 Vizuál objektu *RightWall* v scéne.

Veľkosť objektu *TopWall* nastavíme na hodnotu 18 (16 *units* predstavuje svet a veľkosť pravej a ľavej steny v horizontálnom smere je 1). Pozíciu v smere osi *y* zmeníme na 12,5 a v smere osi *x* do stredu, t. j. na hodnotu 8. Vizuál všetkých stien zachytáva obrázok 124.



Obr. 124 Vizuál objektov *LeftWall*, *RightWall* a *TopWall*.

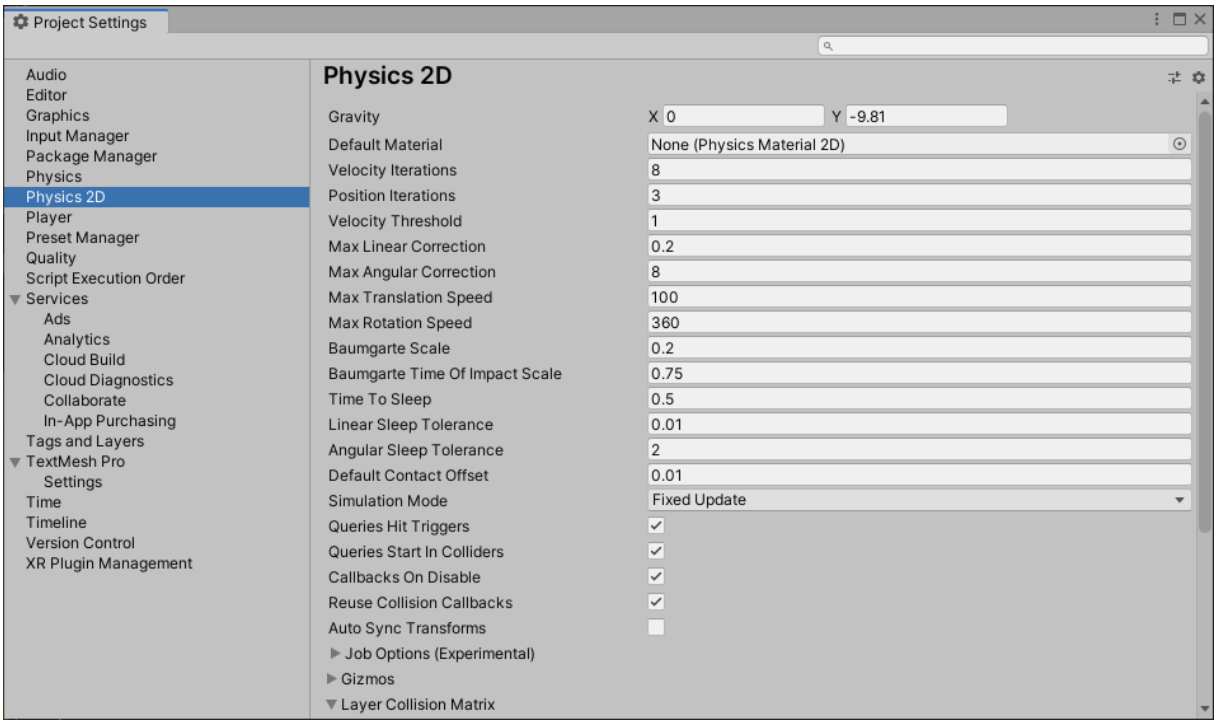
Pozn. autora: Ak by sme veľkosť objektu *TopWall* nastavili na hodnotu 16 – podľa veľkosti herného sveta, mohlo by sa stať, že by sme nezachytili kolíziu v prípade, že by loptička letela presne do stredu ľavého, či pravého rohu. Veľkosť *collideru* v smere *x* pre *LeftWall* a *RightWall*, a v smere *y* pre *TopWall*, ktoré sú momentálne nastavené na hodnotu 1, nie je príliš dôležitá. Detekcia kolízie nastane ihneď pri strete jedného *collideru* s druhým.

6.3 Zmena gravitácie

Ak potrebujeme ovplyvňovať správanie sa objektu vzhľadom k jeho gravitácii, môžeme na to využiť vlastnosť *Gravity Scale*, ktorá sa nachádza v komponente *RigidBody 2D* daného objektu. Východiskové nastavenie predstavuje hodnotu 1. Pri zmene hodnoty napr. na 10 pri objekte *Ball* môžeme sledovať, že loptička by bola príliš ťažká.

Druhou možnosťou, ktorou vieme ovplyvniť gravitáciu všetkých objektov projektu, je pomocou *Edit* → *Project Settings* a kategórie *Physics 2D* (Obr. 125). Ako môžeme vidieť gravitácia je nastavená na hodnotu –9,81 čo odpovedá gravitácii zeme ako

poznáme. Pre našu hru hodnotu nastavíme na 0, čo znamená, že žiadna gravitácia v hre nebude pôsobiť.



Obr. 125 *Physics 2D.*

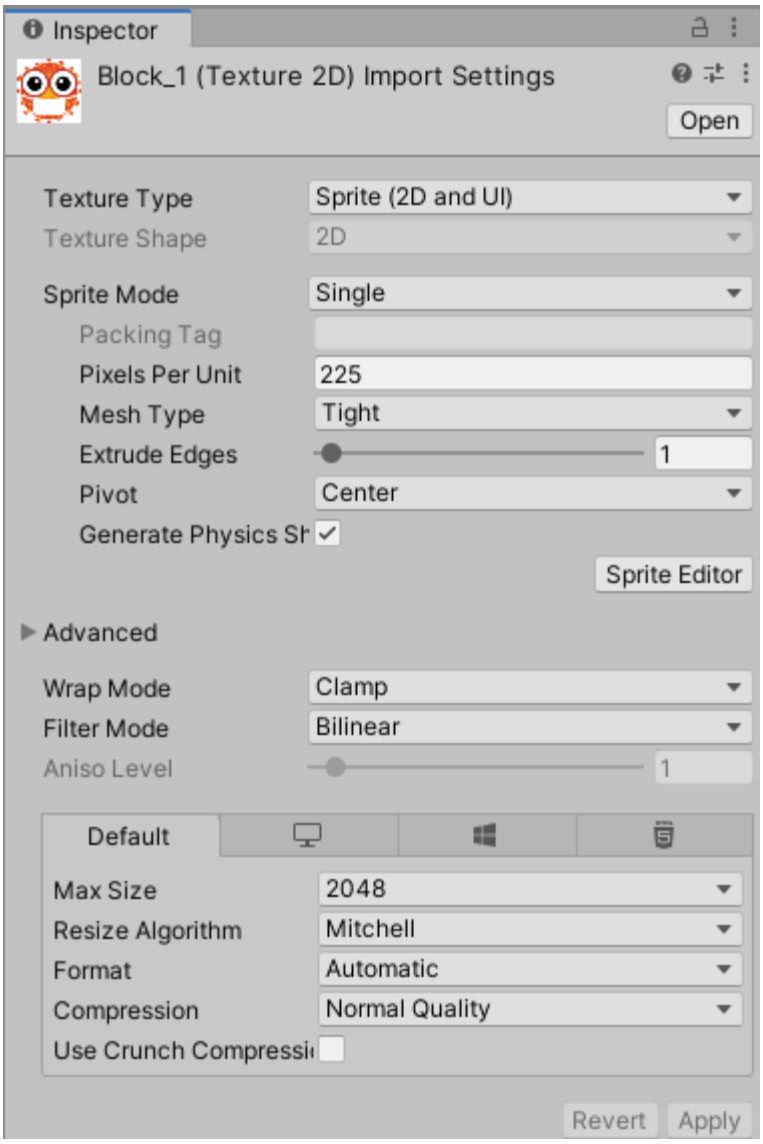
Pozn. autora: Zmena gravitácie pre celý projekt je vzhľadom na želané správanie sa hry vhodnejšia. V prípade ak by sme pridali do projektu ďalšie objekty typu *Ball*, *Block* a pod., gravitácia sa nebude aplikovať ani na ne.

6.4 Kontrolné otázky a úlohy

1. Aké nastavenia je možné ovplyvňovať s pomocou *Script Execution Order*?
2. Ako vieme zabezpečiť funkcionalitu vystrelenia loptičky?
3. K čomu slúži metóda *GetComponent* triedy *GameObject*?
4. Ako vieme ovplyvňovať gravitáciu konkrétneho objektu?
5. Ako vieme ovplyvňovať gravitáciu všetkých objektov projektu?
6. Experimentujte s nastaveniami premenných *xPush* a *yPush* tak, aby ste docielili optimálne správanie sa loptičky.
7. Použite na ohraničenie herného priestoru kruh a sledujte správanie sa loptičky.

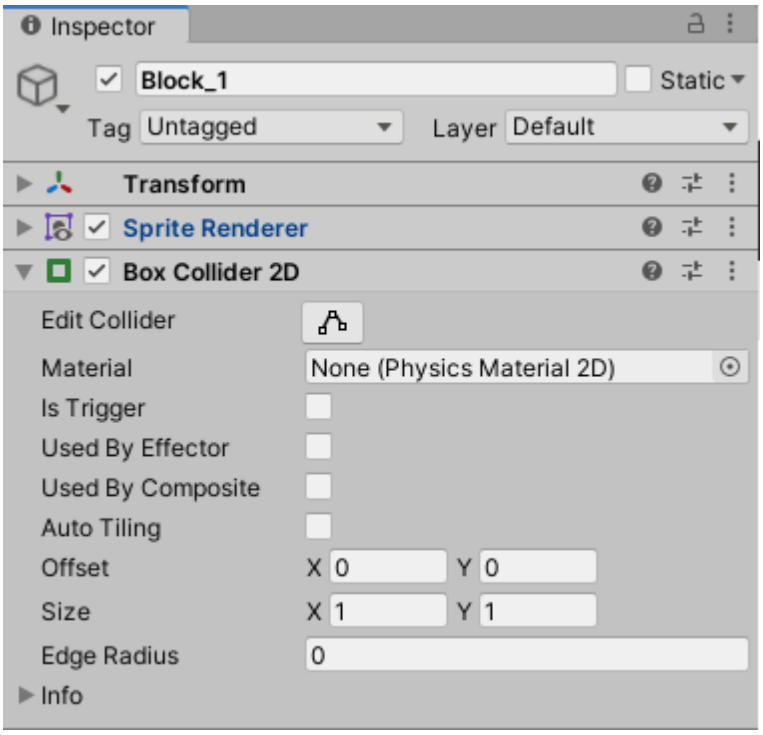
7 Zabezpečenie správania sa objektu *Block*

V tejto časti zabezpečíme zničenie bloku pri náraze loptičky do neho. Vzhľadom na to, že ide o prvý level, a tento by mal byť vzhľadom na hernú logiku jednoduchší ako ďalšie levely, rozhodli sme sa zväčšiť veľkosť bloku dvojnásobne. Aby sa táto zmena aplikovala na všetky bloky použitého typu, robíme túto zmenu v *sprite*. V okne *Inspector* zmeníme hodnoty vlastnosti *Pixel Per Unit* na polovicu, t. j. pôvodnú hodnotu 450 zmeníme na 225 (Obr. 126). Zmeny aplikujeme.



Obr. 126 Zmena veľkosti *spritu Block_1*.

Z dôvodu zmeny veľkosti *spritu Block_1* je potrebné upraviť aj veľkosť *Box Collideru 2D*. Zmenu môžeme zrealizovať pomocou *Edit Collider*, alebo priamo pomocou vlastnosti *Size* v smere *x* ako aj *y* (Obr. 127).



Obr. 127 Zmena veľkosti *collideru* pre objekt *Block_1*.

Funkcionalitu zničenia objektu *Block_1* pri náraze loptičky do neho urobíme pomocou novo vytvoreného skriptu s názvom *Block*, ktorý tomuto objektu pridáme. Funkcionalitu riešime pri kolízii, preto využívame *OnCollisionEnter2D*(), v ktorej zrealizujeme zničenie objektu zavolaním metódy *Destroy()* triedy *Object* (Obr. 128). Metóda *Destroy*() umožňuje zničiť objekt, komponent ale aj *asset*. Metóda má dva parametre, prvý hovorí o tom, čo chceme zničiť, a druhý parameter za koľko to chceme zničiť. V prípade zničenia objektu dôjde k jeho zničeniu aj so všetkými potomkami, ak ich teda objekt má. V našom prípade stačí ako prvý parameter použiť *gameObject*, keďže sa nachádzame v skripte pre objekt *Block*, a zničenie objektu nastavíme napríklad po jednej sekunde.

```
Destroy(gameObject, 1f);
```

V prípade ak chceme aby sa objekt zničil okamžite, volanie bude v tvare:

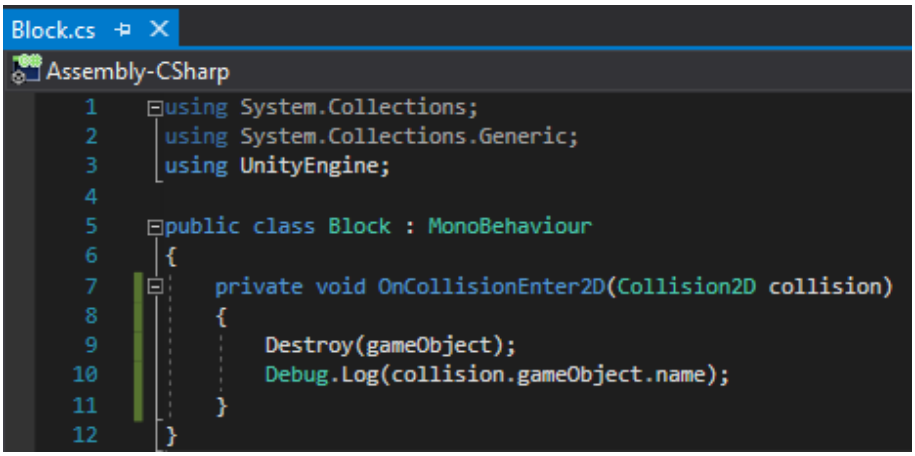
```
Destroy(gameObject);
```

Vďaka tomu, že *onCollisionEnter2D*() má parameter, umožňuje nám to prístup k rôznym informáciám. Napríklad v prípade kolízie si vieme vypísať meno objektu, ktorý kolíziu spôsobil. Ako sme už uviedli vyššie, tieto kontrolné výpisy do konzoly

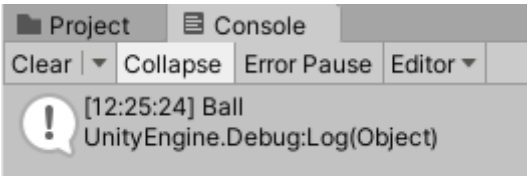
sú užitočné v prípade hľadania chýb, resp. kontroly správania sa implementovanej funkcionality. Ak do *onCollisionEnter2D()* doplníme volanie metódy *Debug()* v tvare:

```
Debug.Log(collision.gameObject.name);
```

môžeme pri testovaní hry v prípade kolízie objektu *Ball* s objektom *Block_1* v konzole sledovať výpis „*Ball*“ (Obr. 129).



Obr. 128 Vizuál skriptu *Block.cs*.



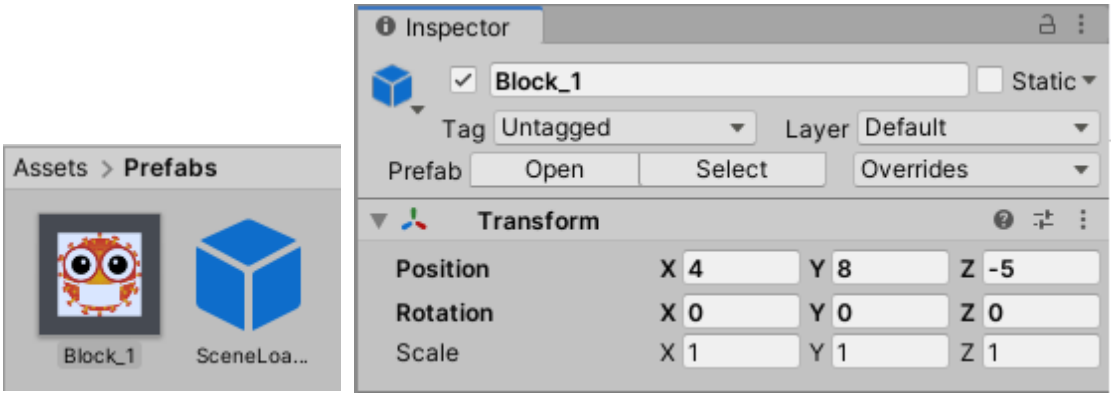
Obr. 129 Výpis v konzole pri kolízii.

7.1 Použitie *prefab* v Unity

Ako sme uviedli v kapitole 2.2.1 v súvislosti s tvorbou objektu *SceneLoader*, *prefab* je niečo ako šablóna (*template*). V tomto prípade ho využijeme v súvislosti s objektom *Block_1*. Je predpoklad, že objektov tohto typu budeme mať v scéne viacero. Ak majú mať tieto objekty rovnaké vlastnosti, je efektívne mať z objektu vytvorený *prefab*, čo znamená, že v prípade akejkoľvek modifikácie budú tieto zmeny rovnako aplikované na všetky objekty tohto typu okrem pozície a rotácie. Tieto sú zvýraznené tučným písmom (*boldom*), čo značí, že sa odlišujú od nastavení *prefabu*, a je teda nutné tieto určiť individuálne pre každú inštanciu (Obr. 130).

Samozrejme, že objekty môžeme vytvárať aj neefektívnym spôsobom a to duplikovaním jednotlivých objektov. Ak by sme však zmenili vlastnosti jedného objektu, napr. by sme zmenili farbu bloku, a chceli by sme, aby sa tieto zmeny aplikovali aj na ostatné objekty, museli by sme tieto zmeny vykonať na každom jednom objekte samostatne. Vďaka používaniu *prefab* sa tomuto vieme vyhnúť.

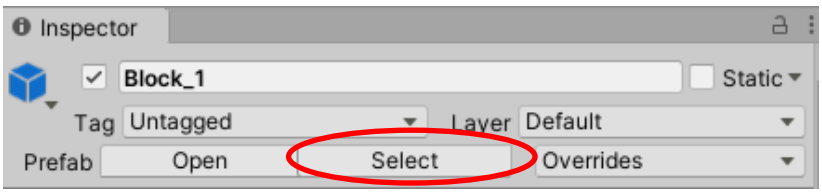
Prefab z objektu *Block_1* vytvoríme jednoducho presunutím tohto objektu z okna *Hierarchy* do priečinku *Prefabs*, ktorý už v *Assets* máme vytvorený.

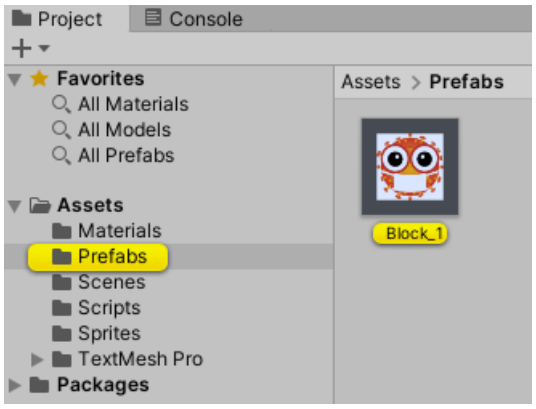


Obr. 130 Vytvorenie *prefabu* z objektu *Block_1*.

Pozn. autora: Súvisiaca terminológia je taká, že ak potiahneme *prefab* do okna *Hierarchy*, resp. priamo do scény, tak vlastne vytvárame inštanciu a tento proces sa nazýva *instantiate*. Tiež sa naučíme ako zabezpečiť vytvorenie inštancie dynamicky pomocou kódu.

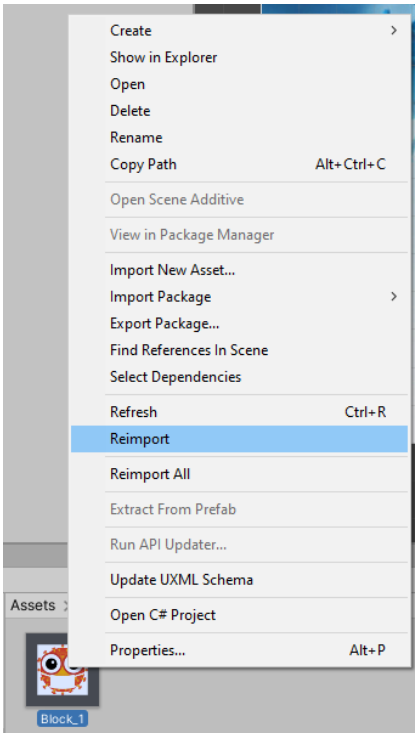
Pri práci s *prefab* sa môžeme stretnúť s nasledovnými možnosťami. Ak máme v scéne vybratý objekt, ktorý je *prefabom*, stlačením tlačidla *Select* v okne *Inspector* nám Unity v našom *Assets* ukáže koho inštanciou tento objekt je:





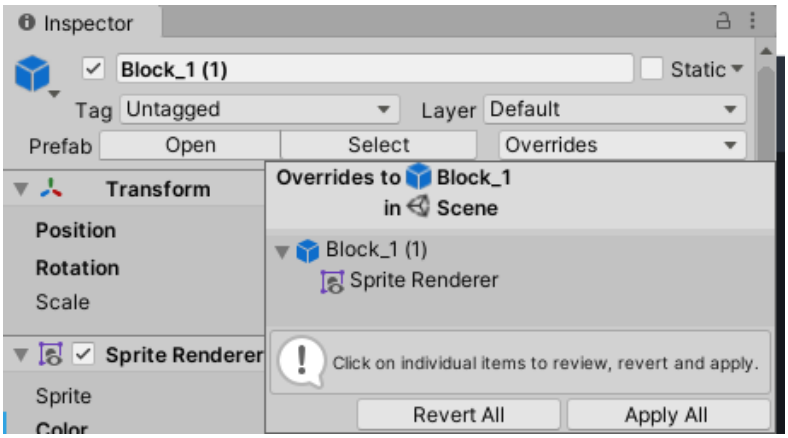
Obr. 131 Zistenie koho inštanciou daný objekt je.

Pozn. autora: Niekedy sa môže stať, že v prípade zmeny vlastností objektu sa tieto zmeny nemusia ihneď v *Assets* prejaviť. Vtedy stačí pomocou kontextovej ponuky použiť možnosť *Reinport*. Samozrejme platí, že ak urobíme nejaké zmeny priamo na *prefabe*, tak sa zmeny prejavia na všetkých inštanciách.



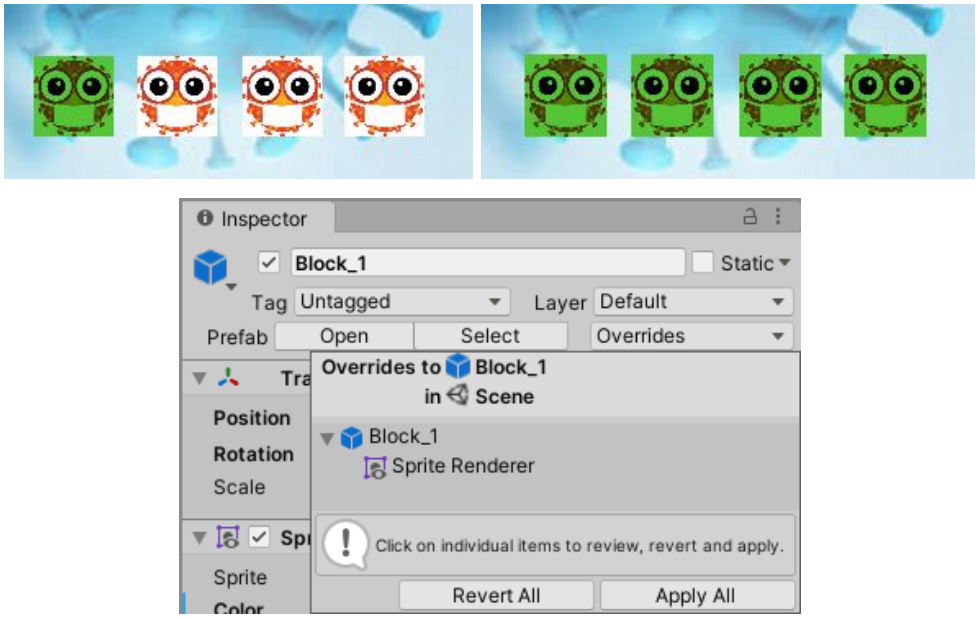
Obr. 132 Vyvolanie možnosti *Reinport*.

Možnosť *Apply All* (Obr. 133) využijeme v prípade zmeny vlastností jedného objektu v scéne s cieľom, aby sa tieto zmeny prejavili na všetkých inštanciách. Napríklad ak zmením farbu jedného objektu a chceme aby sa táto zmena aplikovala na všetky inštancie tohto *prefabu*.



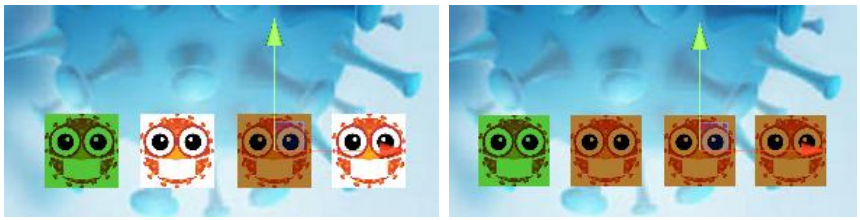
Obr. 133 Možnosť *Apply All*.

Napríklad ak zmeníme farbu jednej inštancie v scéne a chceme aby sa táto zmena aplikovala na všetky ostatné inštancie, ako aj na *prefab*, tak použijeme možnosť *Apply All*.

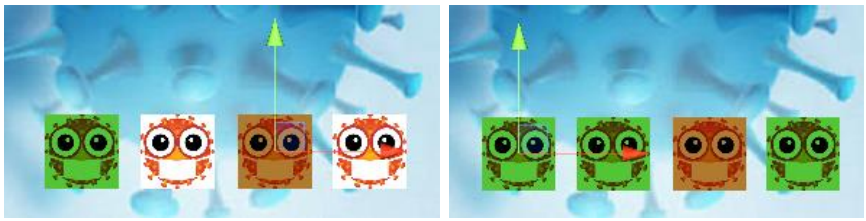


Obr. 134 Zmena farby inštancie a aplikovanie zmien na ostatné inštancie.

V prípade takej situácie, že by jedna inštancia bola zelená a druhá oranžová po manuálnej zmene farby na konkrétnej inštancii a aplikovaním *Apply All* vyvolanej z oranžovej inštancie, sa budú zmeny aplikovať len na tie inštancie, ktoré zodpovedajú *prefabu* (Obr. 135). Ak by zmeny boli vyvolané zo zelenej inštancie, tak by situácia bola taká ako ilustruje obrázok 136.

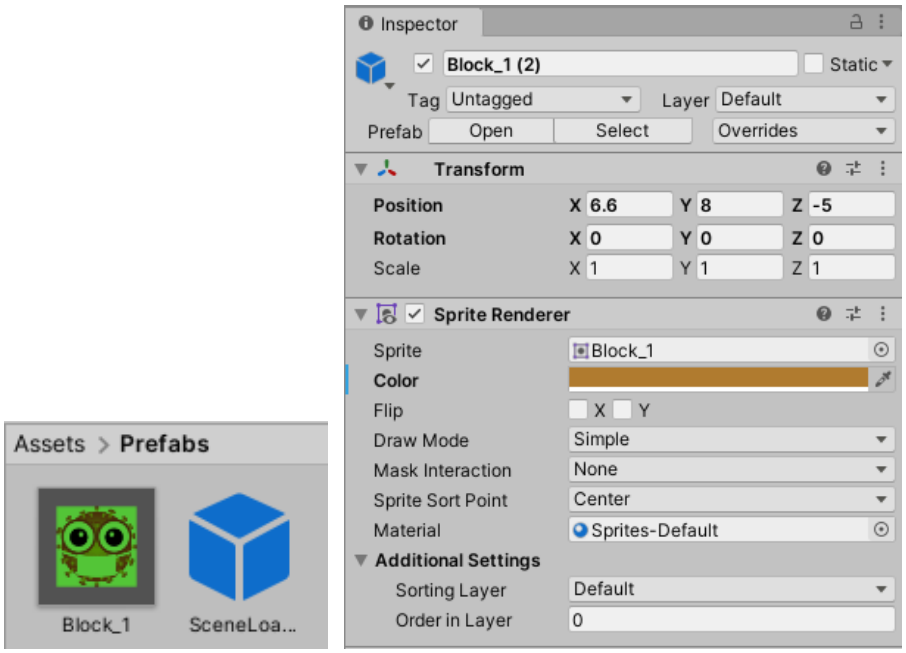


Obr. 135 Aplikovanie *Apply All* z oranžovej inštancie.



Obr. 136 Aplikovanie *Apply All* zo zelenej inštancie.

Na obrázku 137 ilustrujeme skutočnosť, že všetky nastavenia odlišné od *prefabu* sú zvýraznené tučným písmom. V tomto prípade sa líši vlastnosť *Color*. Ako sme uviedli vyššie *Position* a *Rotation* sú vždy iné a preto sú zvýraznené tučným písmom.



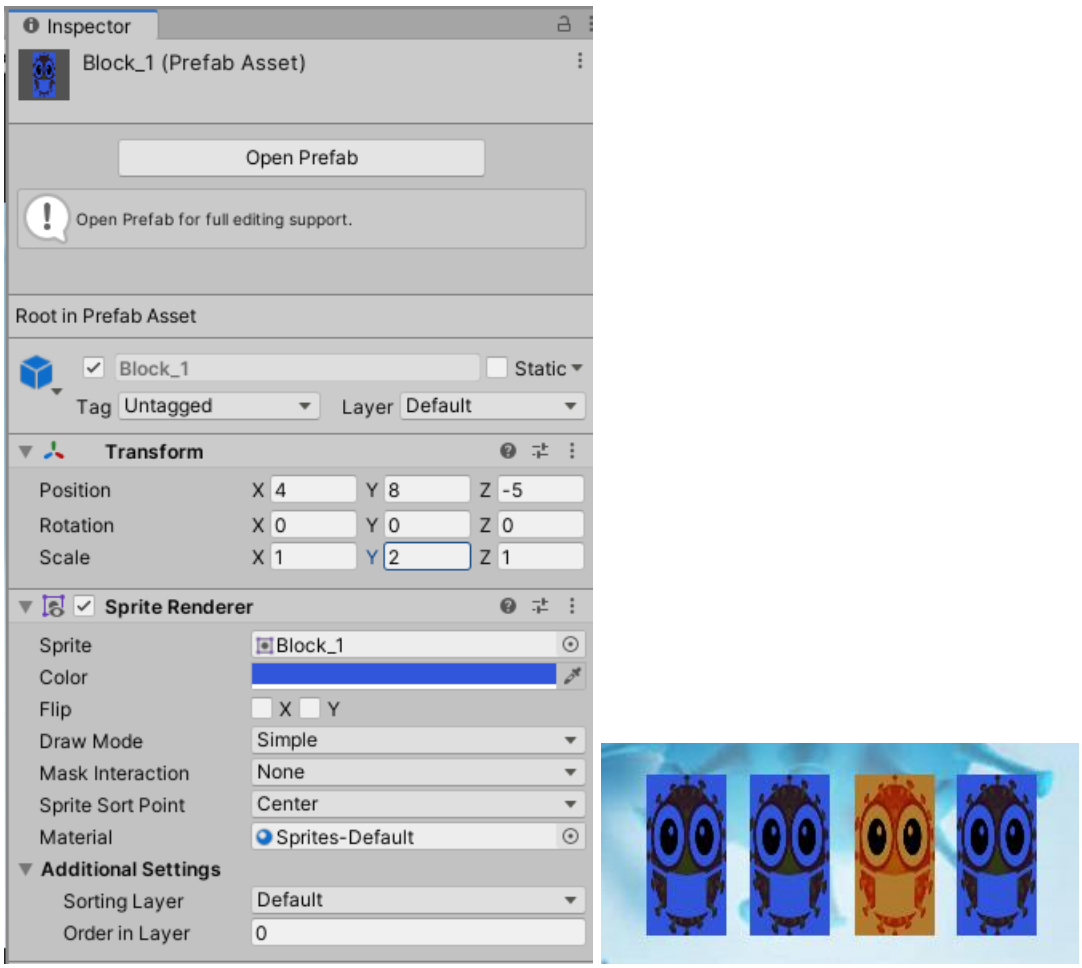
Obr. 137 Ilustrácia odlišných nastavení inštancie od *prefabu*.

V prípade zmeny nastavení na *prefabe* sa tieto aplikujú len na tie inštancie, ktoré sú zhodné s *prefabom*. Napríklad pri zmene farby *prefabu* zo zelenej na modrú sa zmeny prejavia nasledovne:

Obr. 138 Zmena farby *prefabu* a aplikovanie tejto zmeny na inštancie.

To znamená, že oranžová farba ostala, keďže túto sme nastavili priamo na inštancii.

V prípade zmeny vlastnosti *Scale* napríklad v smere osi *y* priamo na prefabe, sa zmeny aplikujú na všetky inštancie bez rozdielu a to z dôvodu, že *Scale* je vlastnosť nezávislá od farby (Obr. 139).



Obr. 139 Zmena vlastnosti *Scale* priamo na *prefabe*.

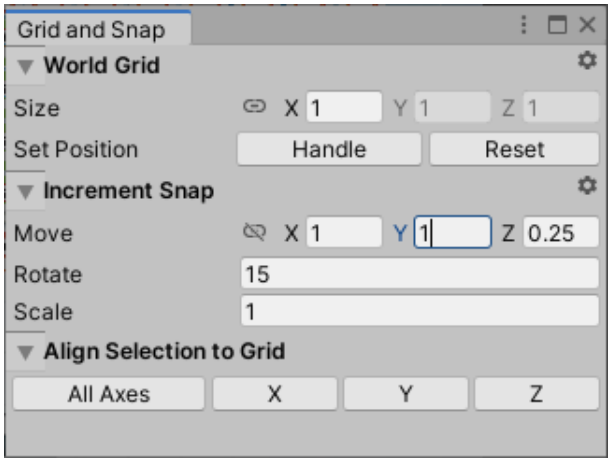
Poslednou možnosťou je *Revert All*, ktorú používame v prípade, ak chceme vrátiť zrealizované zmeny.

7.1.1 *Grid and Snap Settings*

Jednou z veľmi užitočných vecí je, že ak pri pohybe (*move*) nejakého objektu držíme kláves Ctrl, tak sa tento objekt pohybuje a prichytáva podľa aktuálnych nastavení v možnosti *Edit → Grid and Snap Settings*. Aktuálne nastavenia ilustrované obrázkom 140 hovoria o tom, že objekt sa pohybuje v smere osi x a y s krokom 0,25.

Obr. 140 Aktuálne nastavenia *Grid and Snap*.

Vzhľadom na to, že veľkosť jednotlivých blokov je 1, zmenením nastavení v smere osi x a y na hodnotu 1 docielime, že objekty budeme môcť umiestňovať jeden vedľa druhého.

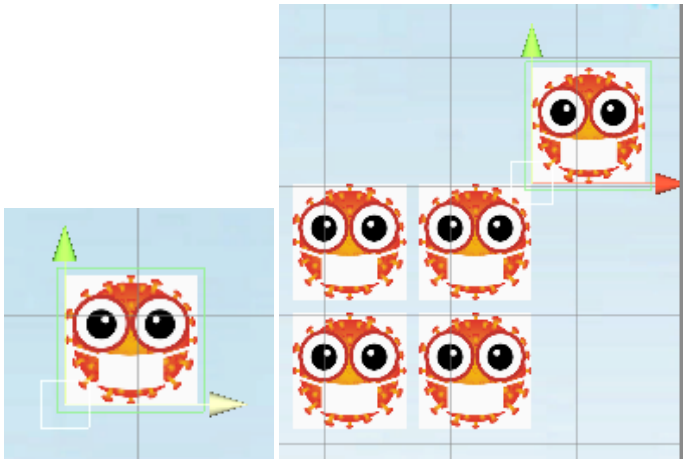


Obr. 141 Zmenené nastavenia *Grid and Snap*.

Pomocou kombinácie kláves Ctrl + D vieme duplikovať objekty, pomocou klávesy Shift vieme označiť viaceré súvislé bloky objektov, resp. pomocou klávesy Ctrl nesúvislé bloky. Ak pri posune držíme kláves Ctrl tak sa pohybujeme tak, ako máme

nastavené prichytávanie. Takto vieme v prípade potreby vytvárať stále väčšie a väčšie zhluky blokov.

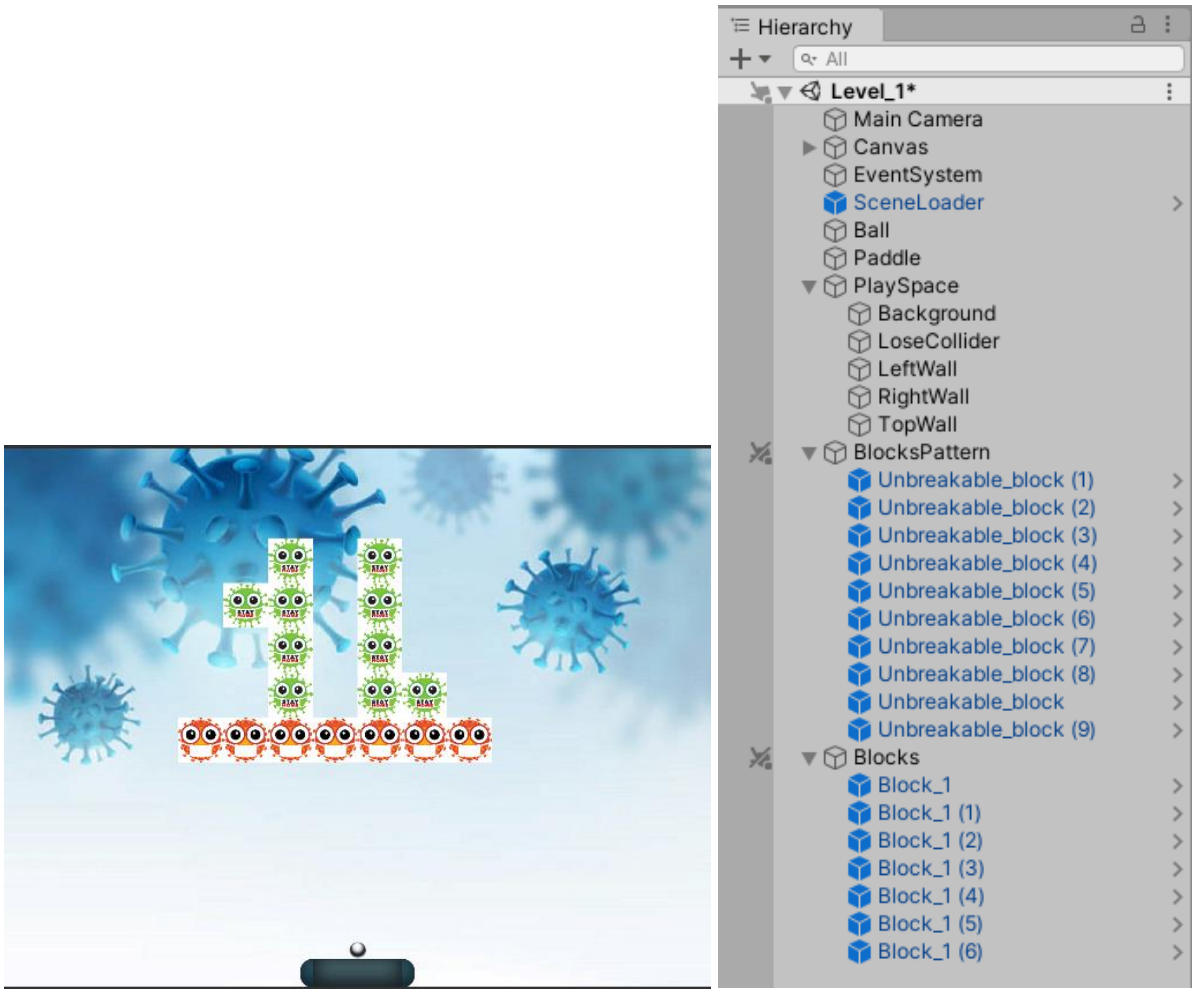
Ďalším užitočným klávesom je kláves V (V = *vertices*), vďaka ktorému sa na objekte zaktívni päť rôznych bodov (rohy + stred), a v prípade pohybu objektom vieme tento prichytávať vzhľadom na vybratý bod. *Obrázok* 142 ilustruje prácu s ľavým dolným rohom.



Obr. 142 Prichytenie objektov s využitím klávesu V.

Mali by sme mať dostatočné vedomosti ako manipulovať s objektmi, aby sme vedeli navrhnúť vizuál prvého levelu. Miera kreativity je na študujúcom. My sme vzhľadom na to, že ide o prvý level zvolili reprezentáciu (Obr. 143), kde je nutné hráčom rozbiť len oranžových „koviďákov“. Zelený „koviďáci“ tvoria len dizajn, preto nie je nutné tomuto objektu pridať skript *Block*.

V okne *Hierarchy* sa snažíme udržiavať poriadok, rovnako ako v celom projekte. Z tohto dôvodu sme vytvorili dva prázdne *Game Objecty*, ktoré sme pomenovali *Blocks* a *BlocksPattern*, kde sme umiestnili vytvorené inštancie blokov (Obr. 144). Objektu *Unbreakable_Block* sme pridali *Box Collider 2D*, aby sme zabezpečili, že ním loptička neprejde, ale skript *Block.cs* sme nepridali, keďže tento slúži len ako grafika daného levelu. Z objektu sme vytvorili *prefab* a vytvorili daný vzor. Pomenovanie jednotlivých inštancií nie je momentálne dôležité. Je zrejmé, že kopírovaním sa aplikuje pravidlo: *Block_1* je pôvodný objekt a *Block_1 (1)* je prvá kópia tohto objektu.



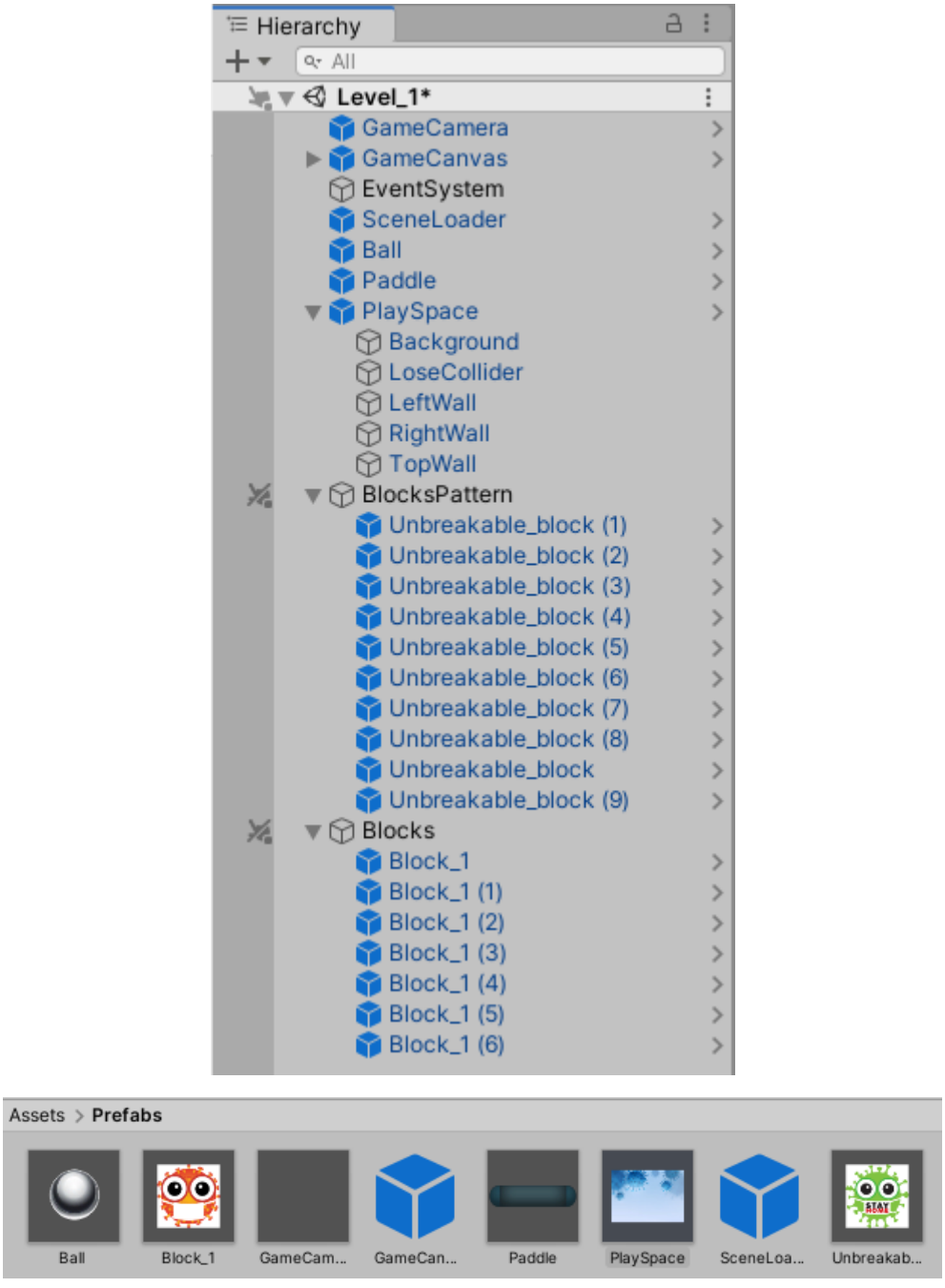
Obr. 143 Vizuál a okno *Hierarchy* prvého levelu.

7.2 Kontrolné otázky a úlohy

1. Akú metódu môžeme použiť pre zničenie objektu? Aký je jej prototyp?
2. Definujte *prefab*.
3. Objasnite pojem *instantiate* a pojem inštancia.
4. Ako je možné aplikovať zrealizované zmeny na *prefabe* na všetky ostatné inštancie?
5. Kde je možné upraviť nastavenia mriežky?
6. Experimentujte s vytvorením dizajnu levelu na základe vlastnej kreativity.

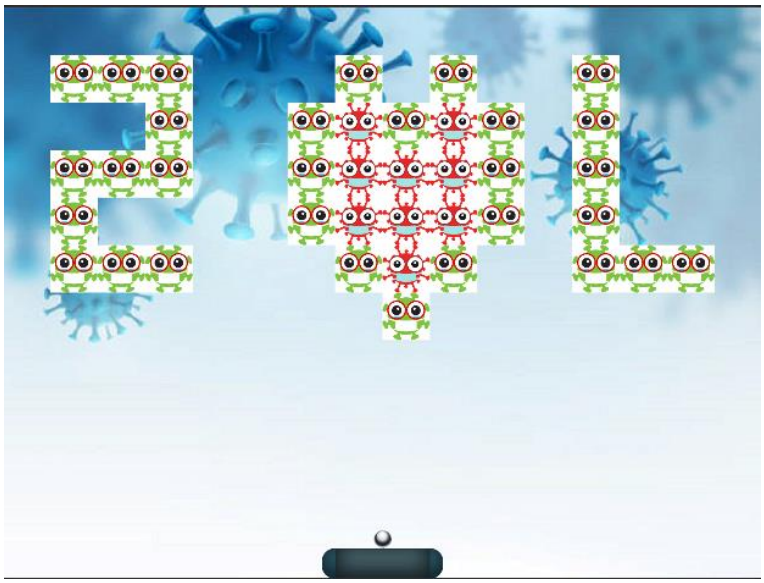
8 Vytvorenie novej scény – Level 2

Ako prvé si z jednotlivých objektov, ktoré máme v scéne *Level_1*vytvoríme *prefaby*, aby sme ich mohli jednoducho použiť pri budovaní ďalšieho levelu. Z dôvodu vyššej výpovednej hodnoty objekt *Main Camera* premenujeme na *GameCamera* a *Canvas* na *GameCanvas*. Všetky objekty v okne *Hierarchy*, ktoré sú modrou farbou sú *prefaby* (Obr. 144).



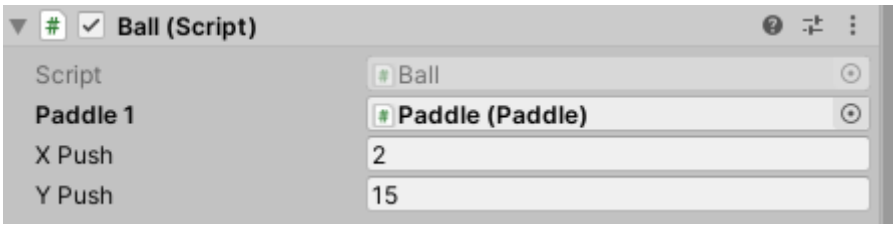
Obr. 144 Vytvorenie *prefabov* z objektov scény *Level_1*.

Novú scénu s názvom *Level_2* vytvoríme duplikovaním Ctrl + D scény *Level_1*. Dizajn levelu ponechávame na čitateľovi, prípadne sa môže inšpirovať vizuálom na obrázku 145.



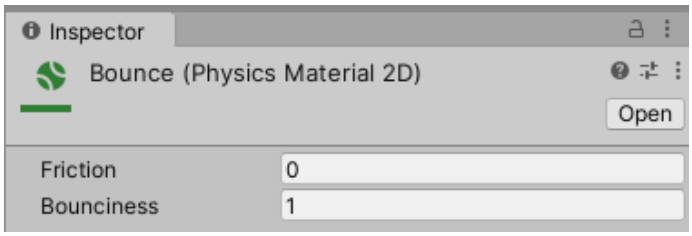
Obr. 145 Vizuál scény *Level_2*.

Pozn. autora: V prípade ak ste sa rozhodli novú scénu vytvoriť manuálne, môže sa stať, že funkcionalita nebude korektne fungovať. Napríklad, že objekt *Ball* sa nepohybuje spolu s objektom *Paddle*. Dôvodom môžu byť chýbajúce inicializácie premenných, v tomto prípade *paddle1* (Obr. 146). Ak ste scénu vytvorili duplikovaním scény *Level_1* tak by malo všetko bezchybne fungovať.



Obr. 146 Inicializácia premenných objektu *Ball*.

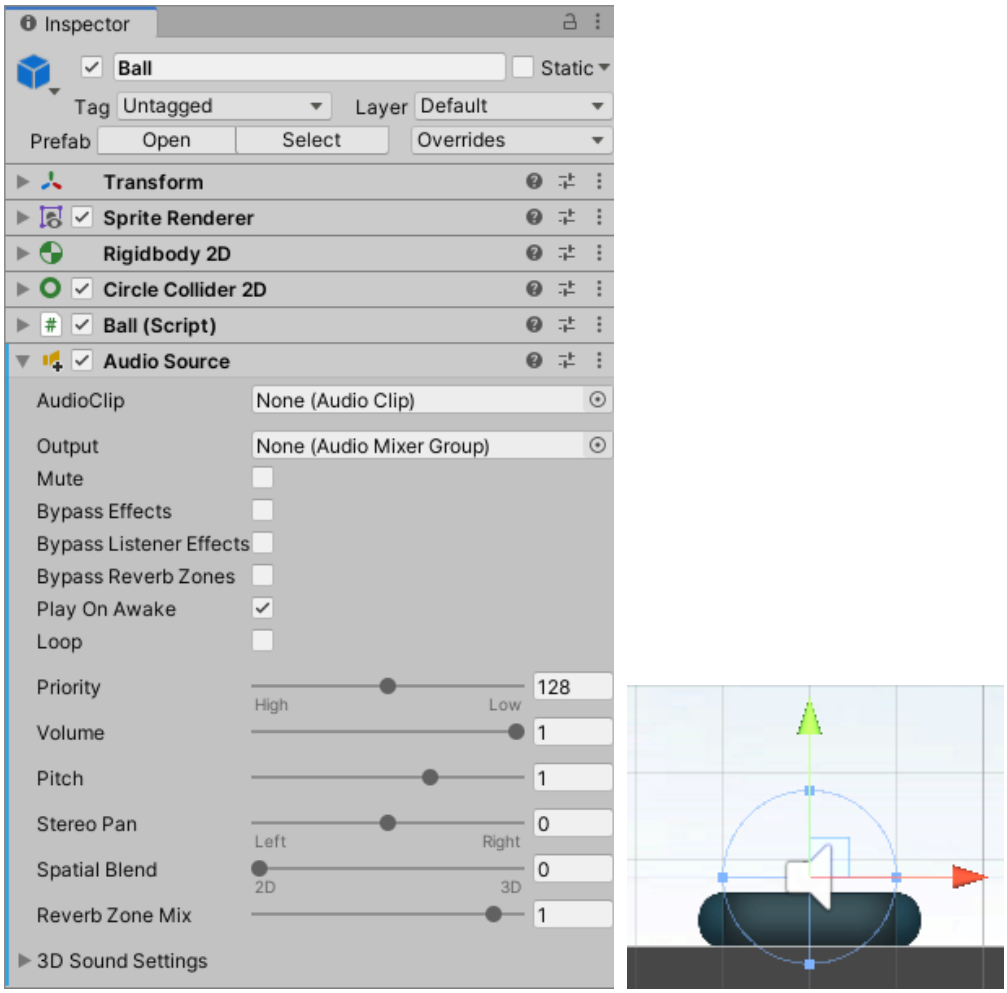
Pri testovaní hry si môžeme všimnúť, že loptička sa pri istých odrazoch (najmä ak loptičku odrazíme hranou padla) spomaľuje. Na objekte *Ball* máme aplikovaný *Physics Material 2D*, ktorý je nastavený v komponente *Circle Collider 2D* vo vlastnosti *Material*. Upravením vlastnosti *Friction* tohto materiálu na hodnotu 0 vyriešime tento nežiaduci jav.



Obr. 147 Upravenie hodnoty vlastnosti *Friction* pre použitý materiál.

8.1 Pridanie zvukových efektov objektu *Ball*

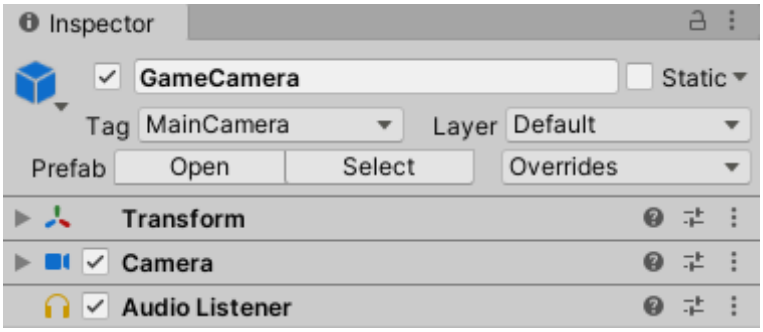
Hra bez použitia zvukových stôp by bola nekompletnou. Zvuk na pozadí, resp. ako efekt pri akciách, je bežnou súčasťou hier. Ak chceme docieliť aby loptička pri náraze do iných objektov vydávala zvuk, pridáme jej komponent *Audio Source*. To, že daný objekt obsahuje takýto komponent je zvýraznené ikonou reproduktora na objekte (Obr. 148).



Obr. 148 Pridanie komponentu *Audio Source* objektu *Ball*.

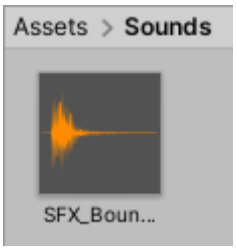
Unity Audio System je flexibilný a silný nástroj, ktorý podporuje prácu s väčšinou štandardných formátov (AIFF, WAV, MP3, Ogg). Po importovaní zvukovej stopy do projektu sa vytvorí *Audio Clip*, ktorý sa pridá objektu, ktorý ho má prehrať. V reálnom živote je zvuk tvorený objektmi (*source*) a počúvaný poslucháčom (*listener*). Obdobne je tomu aj v Unity. Unity dokáže pri zvukoch pracovať s hĺbkou zvuku čo ovplyvňuje jeho hlasitosť, ako aj kvalitu, t. j. záleží od toho odkiaľ zvuk vychádza a kto ho prijíma. Znamená to, že zvuk môže byť produkovaný jedným objektom, v našom prípade objektom *Ball* a prijímať ho bude iný objekt, v našom prípade kamera. V prípade záujmu o simulovanie efektov ako je ozvena (echo) je nutné použiť *Audio Filter*. Ak by sme chceli mixovať rôzne zvukové zdroje a aplikovať rôzne efekty pri tom vám pomôže *Unity Audio Mixer* (Unity, 2020m).

Audio Listener nájdeme na objekte *GameCamera* (Obr. 149) čo zabezpečuje, že môžeme počuť rôzne zvuky produkované objektmi v závislosti od vzdialenosti ako ďaleko sa od nich nachádzame.



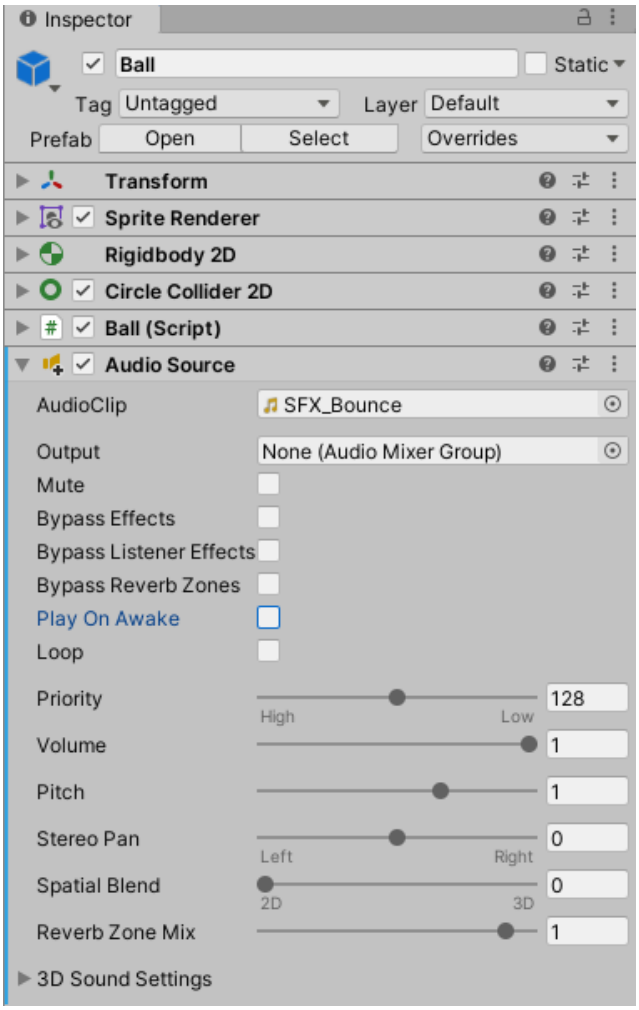
Obr. 149 *Audio Listener* objektu *GameCamera*.

Skôr ako nastavíme zdrojový súbor pre zvuk, je potrebné tento pridať do projektu jednoduchým presunutím súboru do novo vytvoreného priečinku *Sounds*.



Obr. 150 Pridanie zvukového záznamu.

V komponente *Audio Source* vo vlastnosti *AudioClip* zvolíme tento súbor. Vlastnosť *Play On Awake* zabezpečuje, že sa prehrávanie zvuku začne hneď po načítaní scény. Takéto správanie je v tomto momente nežiadúce. Snahou je zabezpečiť, že zvuk sa prehrá v prípade, keď loptička do niečoho narazí. Takže potrebujeme deaktivovať túto možnosť, čo urobíme odškrtnutím *checkboxu* (Obr. 151).



Obr. 151 Nastavenie komponentu *Audio Source* objektu *Ball*.

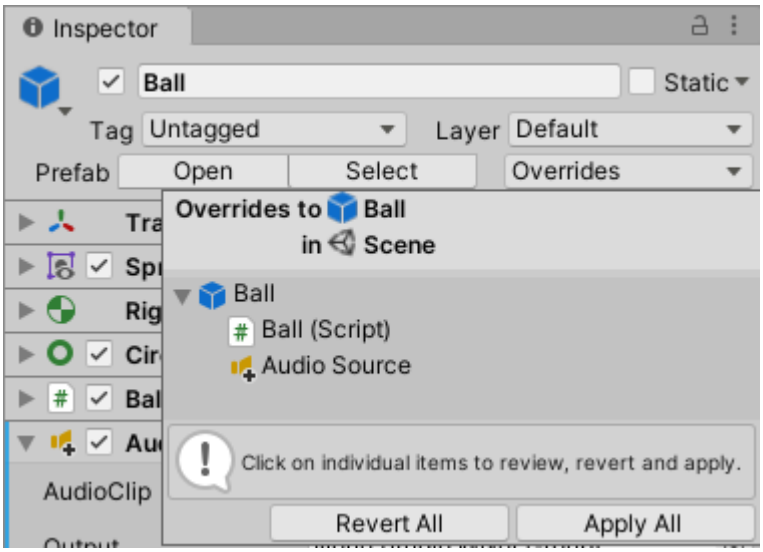
Prehratie zvukovej stopy zabezpečujeme v prípade kolízie objektu *Ball*, preto do skriptu *Ball.cs* doplníme *OnCollisionEnter2D()*. Použijeme v nej metódu *GetComponent()*, pomocou ktorej pristúpime ku komponentu *Audio Source* a zavoláme metódu *Play()* aby nastalo prehratie *Audio Clipu*.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    GetComponent<AudioSource>().Play();
}
```

Ak pozorne testujeme funkčnosť hry, tak počujeme prehratie tohto *clipu* hneď na začiatku, pretože sa deteguje kolízia s objektom *Paddle* (loptička sedí na padle). V prípade ak nepočujeme, môže to byť spôsobené tým, že objekt *Ball* máme umiestnený nad objektom *Paddle*, t. j. k žiadnej kolízii tam nedochádza. Odstránenie tohto nežiaduceho efektu je jednoduché. Stačí využiť premennú *hasStarted*, ktorá nám signalizuje či už bola loptička vystrelená, alebo nie:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if(hasStarted)
    {
        GetComponent<AudioSource>().Play();
    }
}
```

Nezabudneme všetky realizované zmeny aplikovať na *prefab* objektu *Ball* (Obr. 152), t. j. bez ohľadu či sme tieto zmeny robili v scéne *Level_1*, alebo *Level_2* budú tieto aplikované na všetky inštancie objektu Ball nachádzajúce sa v projekte. Ak by sme zmeny neaplikovali, mohlo by byť správanie v leveloch rôzne.



Obr. 152 Aplikovanie zmien *prefabu Ball*.

8.2 Pridanie zvukových efektov objektu *Block*

Prehratie iného zvukového záznamu chceme aj v prípade, ak nastane zničenie objektu typu *Block*. Problém je však v tom, že na rozdiel od objektu *Ball* dochádza k ničeniu (*destroy*) týchto objektov, ak sú zasiahnuté loptičkou, a z toho dôvodu by

sa zničil aj komponent *Audio source*, ktorý by bol k tomuto objektu pridaný, t. j. neprišlo by k prehratiu zvuku ako by sme očakávali. Jedným z riešení by mohlo byť, že by sme zničenie objektu pozastavili (*delay*) do momentu kým by nastalo prehratie zvukového záznamu. K dispozícii však máme vhodnejšie riešenie vďaka metóde *PlayClipAtPoint()*, ktorá dokáže prehrať *audio clip* na konkrétnej pozícii v hernom svete (*world space*).

V skripte *Block.cs* vytvoríme premennú typu *AudioClip*, ktorá bude reprezentovať zvukový záznam:

```
[SerializeField] AudioClip breakSound;
```

Volanie metódy *PlayClipAtPoint()* doplníme do *OnCollisionEnter2D()*, ktorá sa vyvolá v prípade kolízie. Prvým parametrom je zvukový záznam, ktorý sa má prehrať a druhým pozícia vo svete. Pre tento parameter využijeme kameru, aby sme zabezpečili dostatočnú hlasitosť prehratia, keďže kamera je aj *listenerom* (Obr. 153).

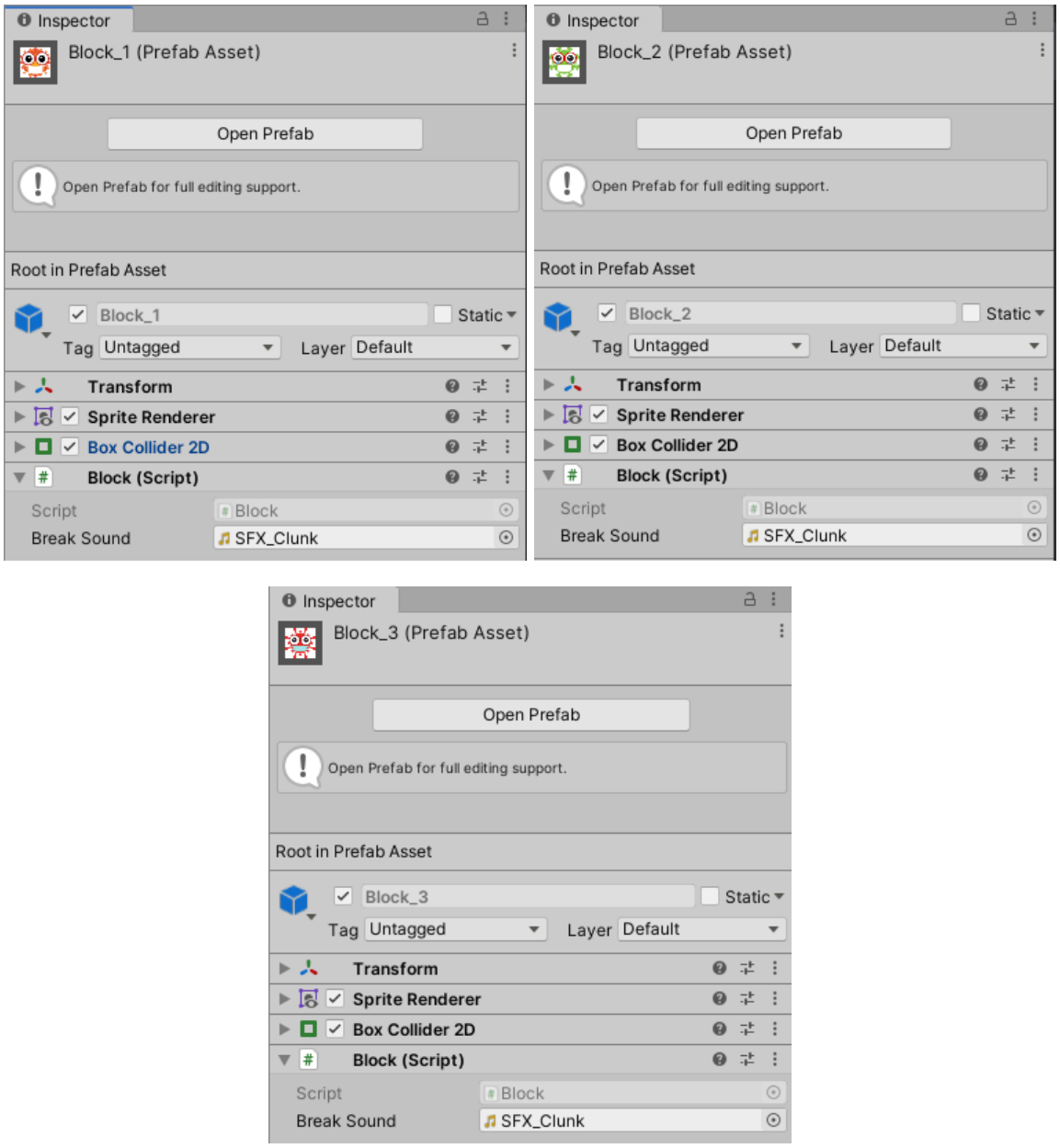
Block.cs

Assembly-CSharp

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Block : MonoBehaviour
6  {
7      [SerializeField] AudioClip breakSound;
8
9      private void OnCollisionEnter2D(Collision2D collision)
10     {
11         AudioSource.PlayClipAtPoint(breakSound, Camera.main.transform.position);
12         Destroy(gameObject);
13         Debug.Log(collision.gameObject.name);
14     }
15 }
```

Obr. 153 Vizuál skriptu *Block.cs*.

V editore Unity nezabudneme inicializovať premennú *breakSound* zvukovým záznamom. Ideálne je tieto zmeny robiť priamo na *prefaboch* blokov, aby sa zmeny aplikovali na všetky inštancie blokov použitých v scénach. My pracujeme v našom projekte už s tromi rôznymi typmi blokov (Obr. 154).



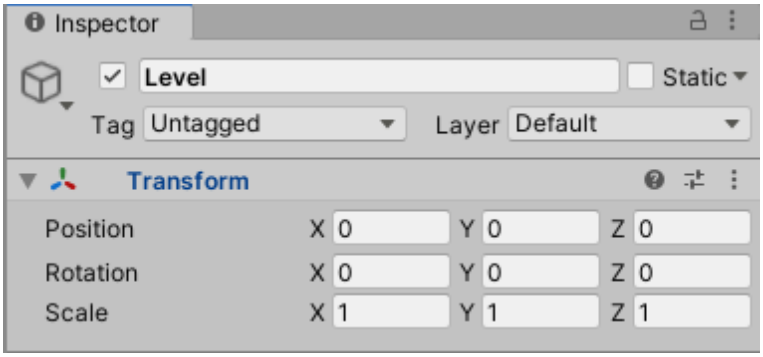
Obr. 154 Inicializácia premennej *breakSound*.

8.3 Ako počítať počet rozbitých blokov

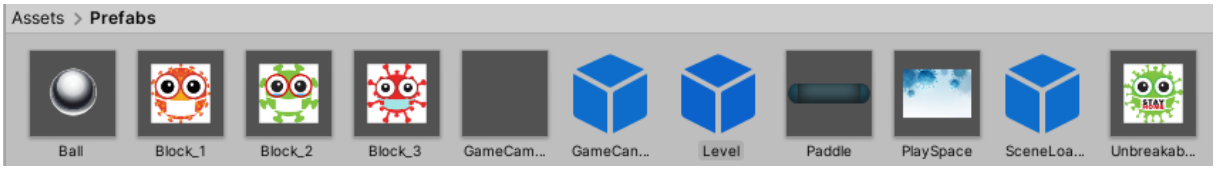
Základný herný princíp hry a prechodu levelu je rozbitie všetkých blokov. Aby sme túto funkcionalitu zabezpečili potrebujeme premennú, ktorá bude uchovávať počet všetkých blokov v danom leveli, ktoré je nutné rozbiť. Potrebovať budeme metódu, ktorá sa bude starať o to, že na začiatku levelu spočíta všetky tieto bloky a metódu, ktorá bude signalizovať že došlo k rozbitiu bloku, zároveň zabezpečí

zničenie bloku a v prípade, ak boli rozbité všetky bloky, zrealizuje načítanie ďalšieho levelu.

Ako prvé vytvoríme prázdny *GameObject*, ktorý pomenujeme *Level* a resetneme jeho nastavenia v *Transform* (Obr. 155). Je jedno či pracujeme v scéne *Level_1* alebo *Level_2*, pretože z tohto objektu vytvoríme *prefab* a pridáme ho do oboch levelov. Ako vieme, *prefab* z objektu vytvoríme jednoduchým presunutím objektu z okna *Hierarchy* do priečinku *Prefabs* (Obr. 156).



Obr. 155 Vytvorenie objektu *Level*.



Obr. 156 Vytvorenie *prefabu* z objektu *Level*.

Objektu pridáme skript s názvom *Level* a vytvoríme v ňom verejnú metódu *CountBreakableBlocks*(), ktorá bude počítať, koľko blokov máme v danom leveli. Vytvoríme si celočíselnú premennú s názvom *breakableBlocks*. Princíp tejto funkcionality bude v tom, že túto metódu vyvoláme vo funkcii *Start*() skriptu *Block.cs*, čím zabezpečíme spočítanie všetkých blokov v leveli, t. j. každý blok započíta sám seba.

```
[SerializeField] int breakableBlocks;

public void CountBreakableBlocks()
{
    breakableBlocks++;
}
```

Pre zabezpečenie prepojenia tried *Level* a *Block* existujú dve možnosti. Prvou možnosťou je, že v skripte *Block.cs* by sme vytvorili premennú *level*, typu *Level*, aby

sme prepojili každý blok s našim skriptom a mohli volať metódu *CountBreakableBlocks*(). Pri tomto spôsobe by sme však museli každému bloku v scéne ručne priradiť, že náš *GameObject Level* reprezentuje túto premennú.

```
[SerializeField] Level;
```

Efektívnejší spôsob spočíva v tom, že si vytvoríme referenciu na triedu *Level* a s využitím metódy *FindObjectOfType*() triedy *Object* vytvoríme spojenie. Táto metóda vráti prvý aktívny objekt daného typu. V prípade ak sa žiadna zhoda nenájde, vráti *null*. V skripte *Block.cs* vytvoríme premennú *level* typu *Level*. V *Start*() vytvoríme spojenie a následne zavoláme metódu *CountBreakableBlocks*() triedy *Level*.

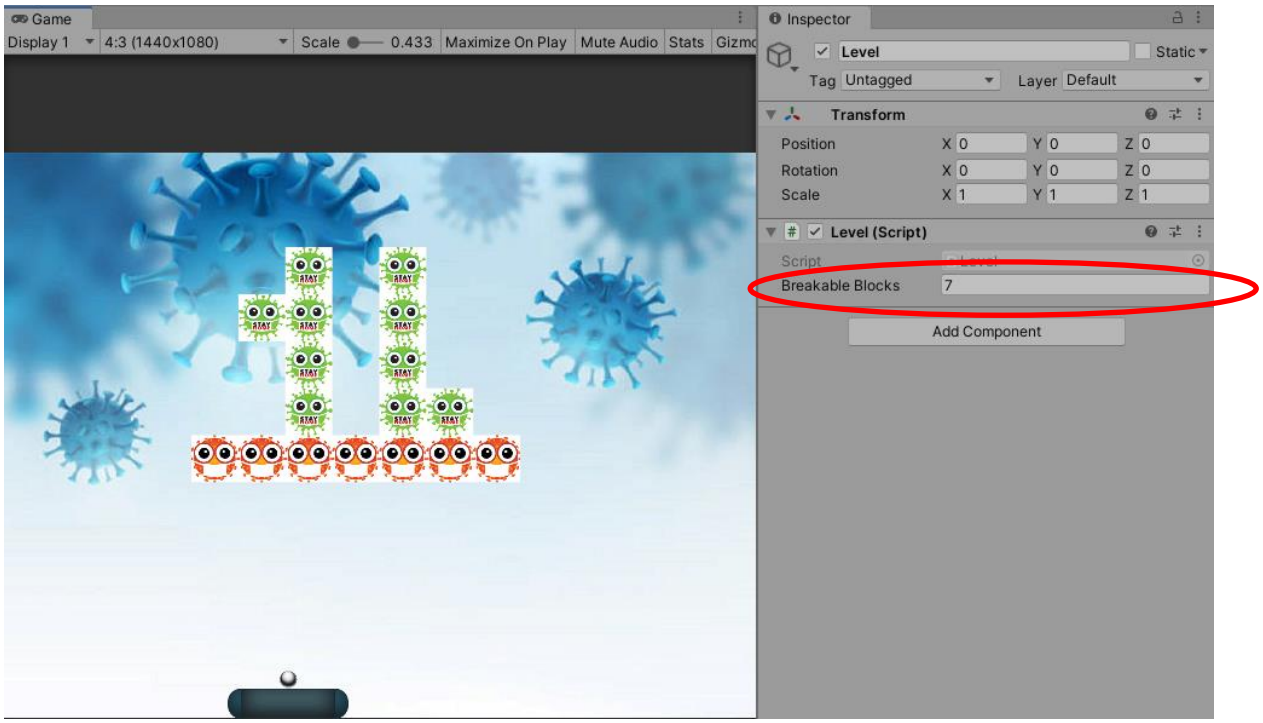
Block.cs*

Assembly-CSharp

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Block : MonoBehaviour
6  {
7      [SerializeField] AudioClip breakSound;
8
9      //cached reference
10     Level level;
11
12     private void Start()
13     {
14         level = FindObjectOfType<Level>();
15         level.CountBreakableBlocks();
16     }
17
18     private void OnCollisionEnter2D(Collision2D collision)
19     {
20         AudioSource.PlayClipAtPoint(breakSound, Camera.main.transform.position);
21         Destroy(gameObject);
22         Debug.Log(collision.gameObject.name);
23     }
24 }
```

Obr. 157 Vizuál skriptu *Ball.cs*.

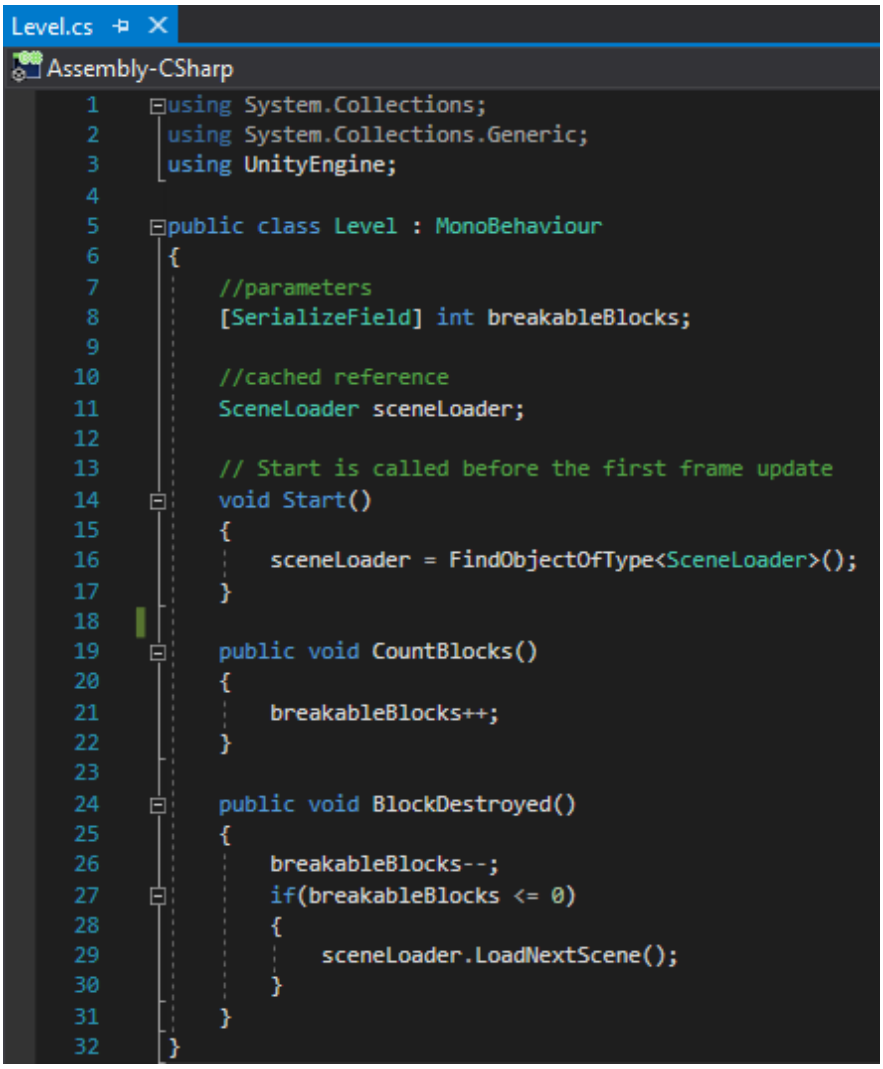
Kontrolu správnej funkcionality je možné vykonať priamo v editore Unity, pri spustení hry kontrolou hodnoty v premennej *breakableBlocks*. Táto reprezentuje počet blokov nutných k rozbitiu.



Obr. 158 Počet blokov k rozbitiu v *Level_1*.

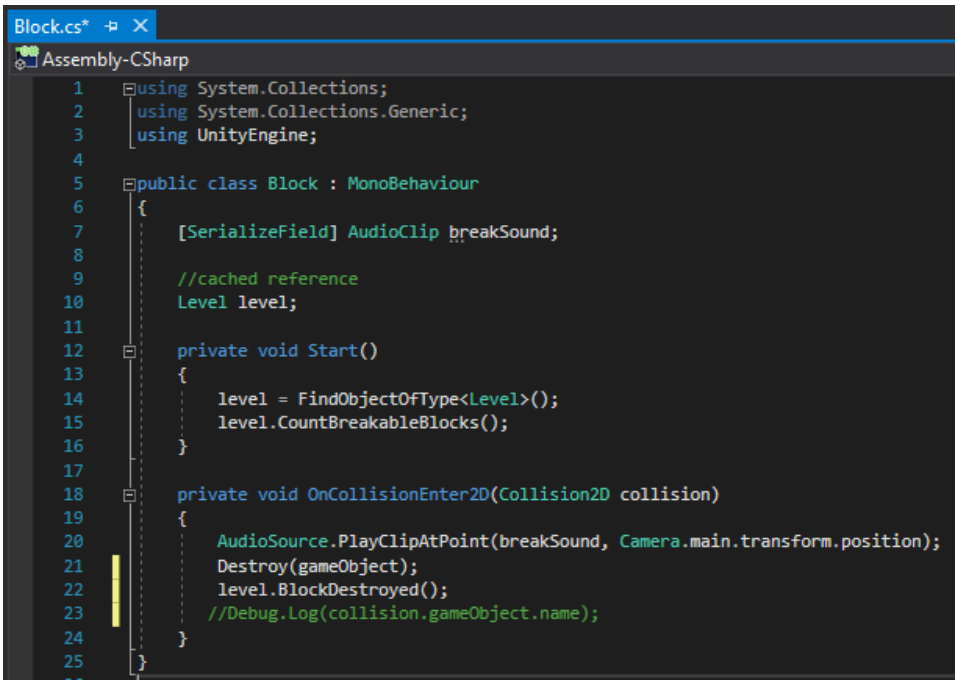
8.3.1 Posun na ďalší level po rozbití všetkých blokov

Cieľom je po odstránení všetkých blokov, ktoré sa nachádzajú v danej scéne posunúť sa na ďalšiu scénu. Vytvárame novú metódu *BlockDestroyed*(), v ktorej vždy odpočítame jeden blok a kontrolujeme či náhodou nie sú všetky bloky zničené. Ak áno, načítame ďalší level zavolaním metódy *LoadNextScene*(). Aby sme mohli zavolať túto metódu, deklarujeme novú premennú *sceneLoader* typu *SceneLoader* a vo funkcii *Start*() s ňou nadviažeme spojenie (Obr. 159).



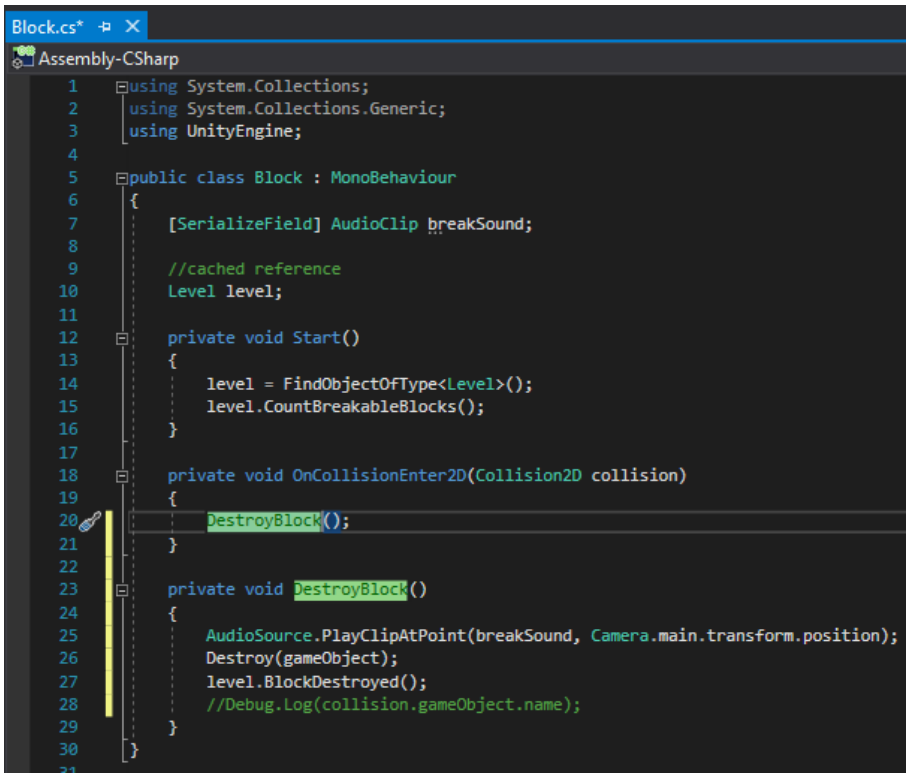
Obr. 159 Vizuál skriptu *Level.cs*.

Metódu *BlockDestroyed()* voláme vždy keď príde k zničeniu bloku, t. j. do skriptu *Block.cs* na riadok 22 doplníme volanie tejto metódy (Obr. 160).



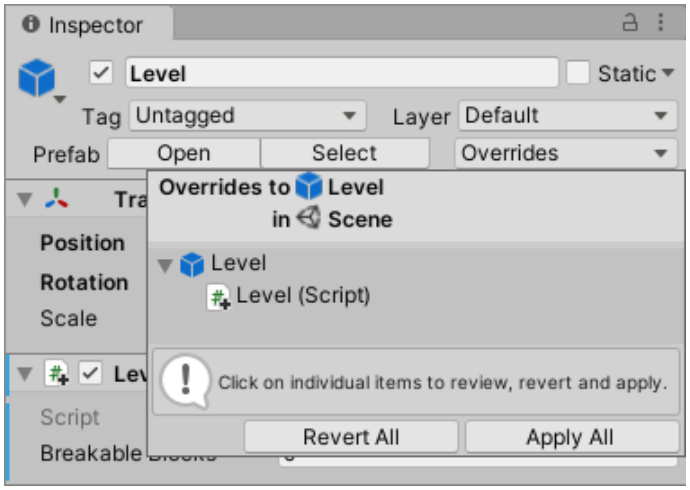
Obr. 160 Vizuál skriptu *Block.cs*.

Pomocou možnosti *Quick Action and Refactoring* a *Extract Method* si časť kódu v *OnCollisionEnter2D()* zapuzdríme do novej metódy s názvom *DestroyBlock()*, z dôvodu väčšej prehľadnosti (Obr. 161).

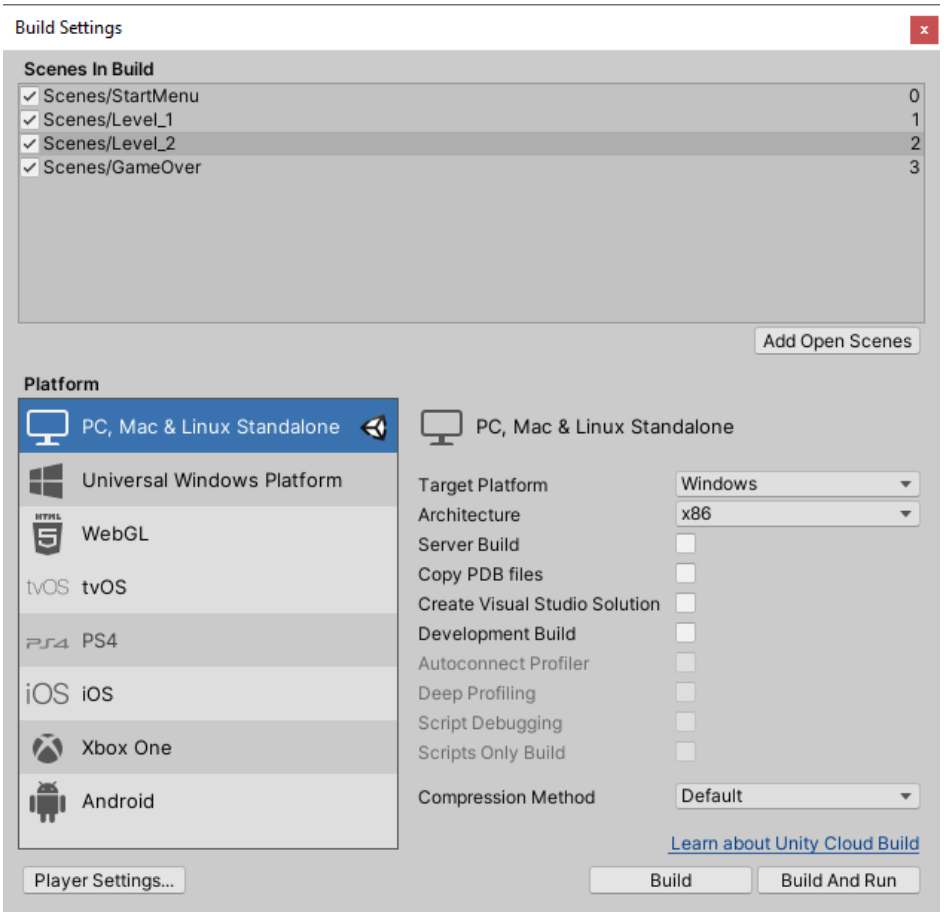


Obr. 161 Vizuál skriptu *Block.cs*.

Pred testovaním funkcionality v editore Unity a v prípade ak chceme aby sa všetky zmeny aplikovali aj na *prefab*, potvrdíme možnosť *Apply All* (Obr. 162). Teraz je možné daný *prefab* pridať aj do druhej scény. Tiež je nutné upraviť nastavenia v *Build Settings*, kde potrebujeme pridať scénu Level_2 (Obr. 163).



Obr. 162 Aplikovanie zmien na *prefab*.

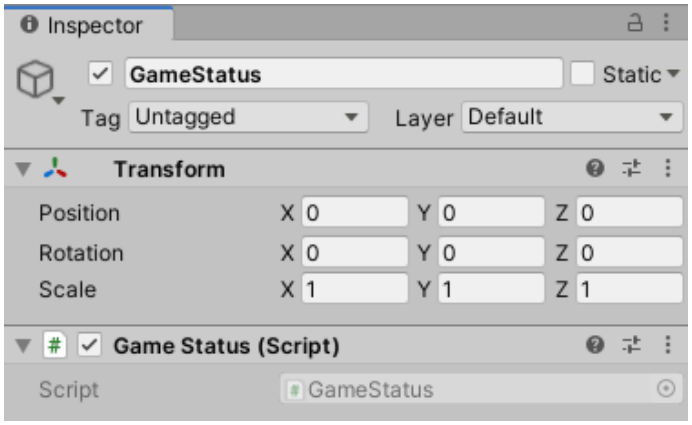


Obr. 163 Úprava nastavení v *Build Settings*.

8.3.2 Úprava rýchlosti hry

Našim cieľom je umožniť modifikáciu rýchlosti hry, s čím nám pomôže vlastnosť *timeScale* triedy *Time*. Vďaka tejto vlastnosti vieme spomaliť efekty pohybu, ako aj zrýchliť aplikáciu. V prípade ak je nastavená na hodnotu 1, čas plynie rovnako ako v reálnom svete. Ak hodnotu upravíme na hodnotu 0,5, tak čas plynie dvakrát pomalšie ako v reálnom svete. Nastavenie na hodnotu 0 spôsobí, že sa aplikácia pozastaví, v prípade ak sú všetky funkcie nezávislé od snímkovej frekvencie (*frame rate independent*). Záporné hodnoty sa ignorujú (Unity, 2020n).

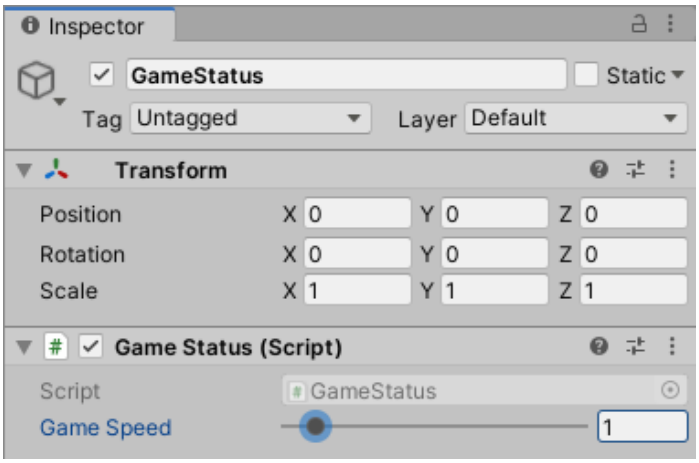
Ako prvé vytvoríme prázdny *Game Object* s názvom *GameStatus*, resetneme jeho nastavenia v *Transform* a priradíme mu skript s rovnakým názvom (Obr. 164). Tento objekt budeme v budúcnosti používať aj na zaznamenávanie skóre.



Obr. 164 Vytvorenie objektu *GameStatus*.

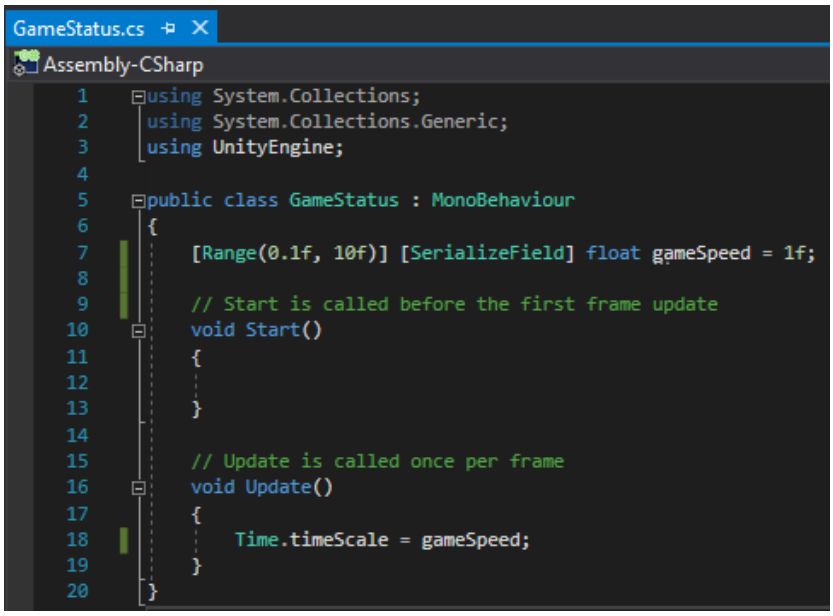
Ak chceme aby sme hodnoty vlastnosti *timeScale* mohli nastavovať len v preddefinovanom rozsahu, môžeme si pomôcť vytvorením *slideru*. Rozsah sme zvolili od 0 ÷ 10:

```
[Range(0.1f, 10f)][SerializeField] float gameSpeed = 1f;
```



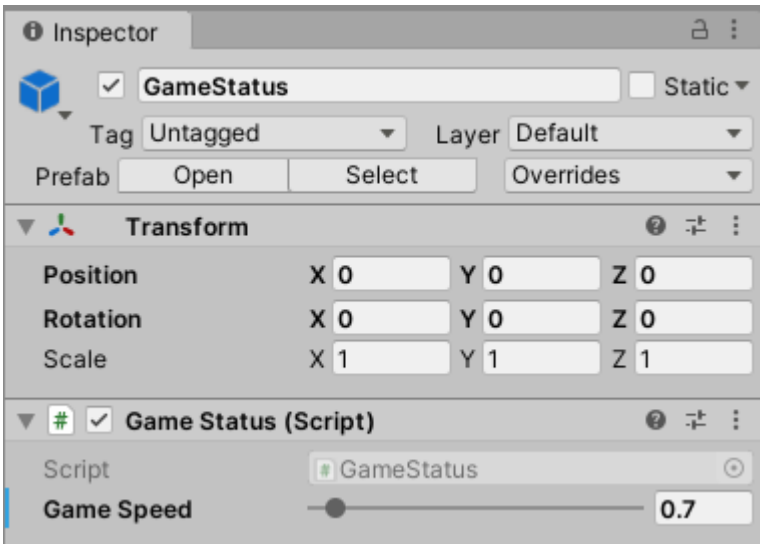
Obr. 165 Výsledok pridania slideru v editore Unity.

Momentálne nastavenie ponechávame na hodnote 1. V budúcnosti môžeme použiť túto funkcionalitu na zmenu rýchlosti reakcie loptičky a pod. Nezabudneme nastaviť hodnotu vlastnosti pomocou vytvorenej premennej *gameSpeed* vo funkcii *Update*() (Obr. 166).



Obr. 166 Vizuál skriptu *GameStatus.cs*.

Z objektu *GameStatus* vytvoríme *prefab*, pretože ho použijeme v oboch scénach, ktoré predstavujú level. V scéne *Level_1* upravíme nastavenie pre *gameSpeed* na hodnotu 0,7 z dôvodu, že ide o prvý level. Tak by mala byť hrateľnosť hry o niečo jednoduchšia ako v druhom leveli. Ak zmeny neaplikujeme na *prefab*, tieto ostanú jedinečné pre konkrétnu inštanciu, o čom hovorí aj zvýraznenie tejto vlastnosti tučným písmom (Obr. 167).



Obr. 167 Úprava hodnoty premennej *gameSpeed*.

8.4 Pridanie skóre do hry

Čo by to bolo za hru, keby sme nepočítali a hráčovi nezobrazovali skóre. Cieľom je pri zničení bloku navýšiť skóre o toľko bodov koľko si určíme. Zadefinujeme si v skripte *GameStatus.cs* dve nové premenné:

- *pointsPerBlockDestroyed* – predstavuje hodnotu, ktorá sa priráta pri zničení bloku,
- *currentScore* – predstavuje aktuálne skóre, ktoré inicializujeme na hodnotu 0.

Vytvoríme metódu *AddToScore()*, ktorá bude aktualizovať skóre o hodnotu nastavenú v premennej *pointsPerBlockDestroyed* (Obr. 168).

GameStatus.cs

Assembly-CSharp

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class GameStatus : MonoBehaviour
6   {
7       //config params
8       [Range(0.1f, 10f)] [SerializeField] float gameSpeed = 1f;
9       [SerializeField] int pointsPerBlockDestroyed = 2;
10
11      //state variables
12      [SerializeField] int currentScore = 0;
13
14      // Start is called before the first frame update
15      void Start()
16      {
17
18      }
19
20      // Update is called once per frame
21      void Update()
22      {
23          Time.timeScale = gameSpeed;
24      }
25
26      public void AddToScore()
27      {
28          currentScore += pointsPerBlockDestroyed;
29      }
30  }
```

Obr. 168 Vizuál skriptu *GameStatus.cs*.

V skripte *Block.cs* v časti kde dochádza ku kolízii loptičky s blokom zavoláme túto metódu – riadok 28 obrázok 169.

```
FindObjectOfType<GameStatus>().AddToScore();
```

Block.cs*

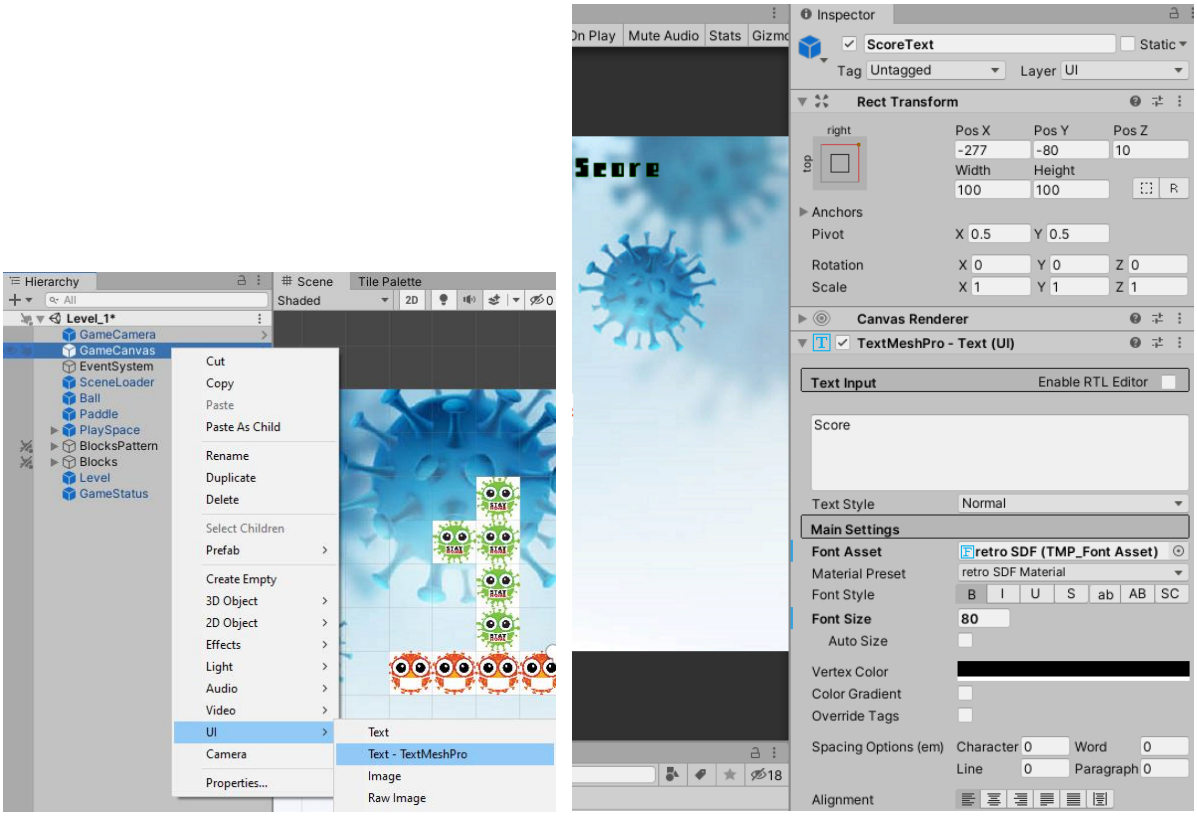
Assembly-CSharp

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Block : MonoBehaviour
6   {
7       [SerializeField] AudioClip breakSound;
8
9       //cached reference
10      Level level;
11
12      private void Start()
13      {
14          level = FindObjectOfType<Level>();
15          level.CountBreakableBlocks();
16      }
17
18      private void OnCollisionEnter2D(Collision2D collision)
19      {
20          DestroyBlock();
21      }
22
23      private void DestroyBlock()
24      {
25          AudioSource.PlayClipAtPoint(breakSound, Camera.main.transform.position);
26          Destroy(gameObject);
27          level.BlockDestroyed();
28          FindObjectOfType<GameStatus>().AddToScore();
29          //Debug.Log(collision.gameObject.name);
30      }
31  }
```

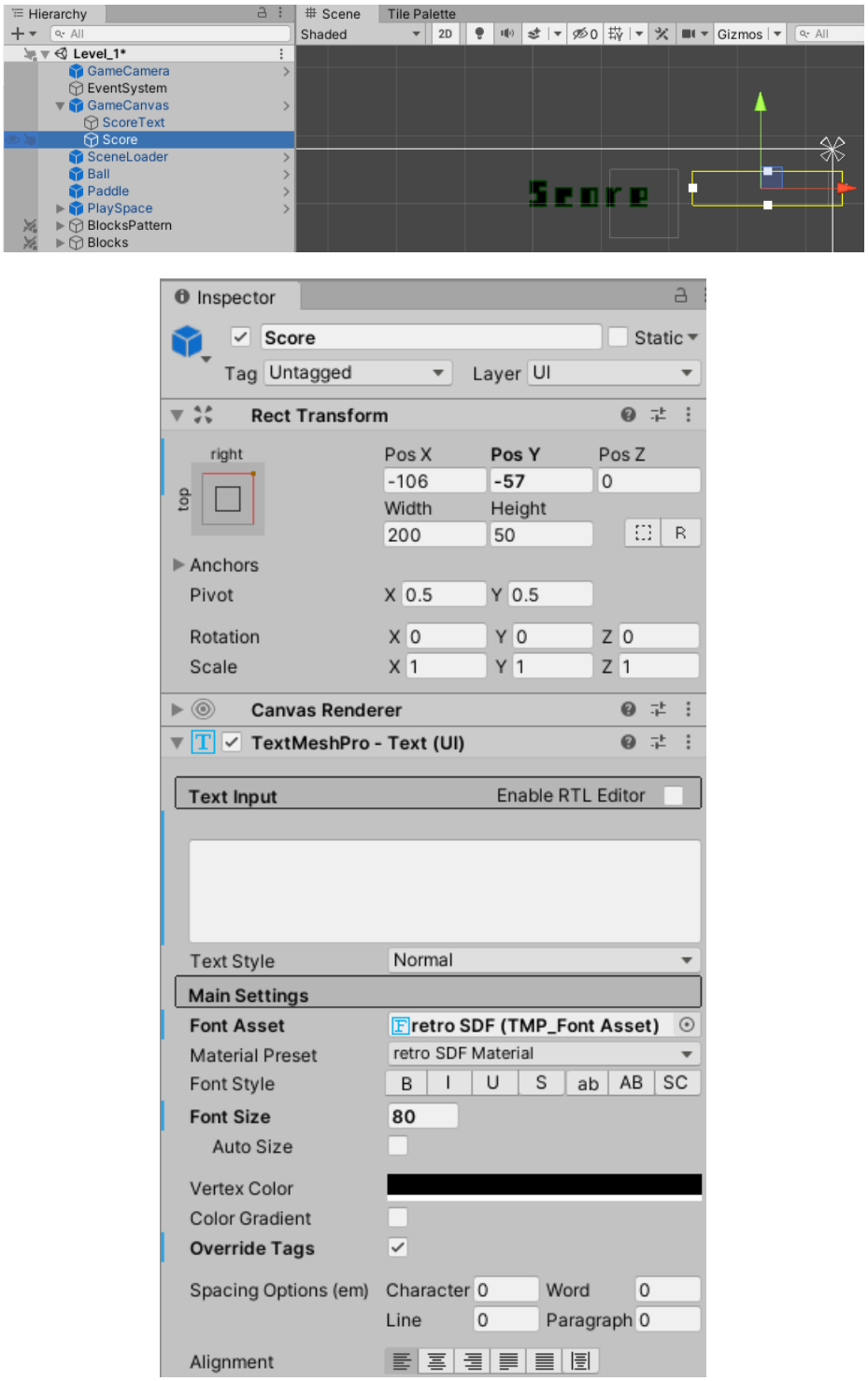
Obr. 169 Vizuál skriptu *Block.cs*.

8.4.1 Zobrazenie skóre v hre

V objekte *GameCanvas* môžeme odstrániť objekt *NextButton*, ktorý už nebudeme potrebovať z dôvodu, že sme implementovali herný mechanizmus prechodu medzi levelmi na základe rozbitia všetkých blokov. Pridáme však dva nové objekty kategórie *UI TextMeshPro*, ktoré pomenujeme *ScoreText* a *Score*. Objekty umiestnime do pravého horného rohu a tiež ich ukotvíme, nastavíme font a veľkosť podľa preferencií (Obr. 170, 171). Použili sme nový font *Retro* stiahnutý z webu https://www.dafont.com/. Objekt *ScoreText* slúži na zobrazenie reťazca „*Score*“ a objekt *Score* bude zobrazovať aktuálnu hodnotu nahratého skóre hráčom.



Obr. 170 Pridanie objektu *ScoreText* a jeho nastavenia.



Obr. 171Pridanie objektu *Score* a jeho nastavenia.

Teraz sa postaráme o zobrazenie skóre na obrazovke. Ako prvé pridáme do skriptu *GameStatus.cs namespace* [TMPro](#) vďaka ktorému budeme môcť využívať metódy triedy [*TextMeshProUGUI*](#).

```
using TMPro;
```

Vytvoríme premennú *scoreText* typu *TextMeshProUGUI*.

```
[SerializeField] TextMeshProUGUI scoreText;
```

Vo funkcii *Start()* zabezpečíme zobrazenie obsahu premennej *currentScore* v objekte *Score* vďaka vlastnosti *text* tohto objektu. Vzhľadom na to, že premenná *currentScore* je celočíselná premenná, používame konverznú funkciu *ToString()*.

```
void Start()
{
    scoreText.text = currentScore.ToString();
}
```

Túto hodnotu je nutné aktualizovať v prípade navýšenia skóre, preto musíme tento príkaz doplniť aj do metódy *AddToScore()*:

```
public void AddToScore()
    {
        currentScore += pointsPerBlockDestroyed;
        scoreText.text = currentScore.ToString();
    }
```

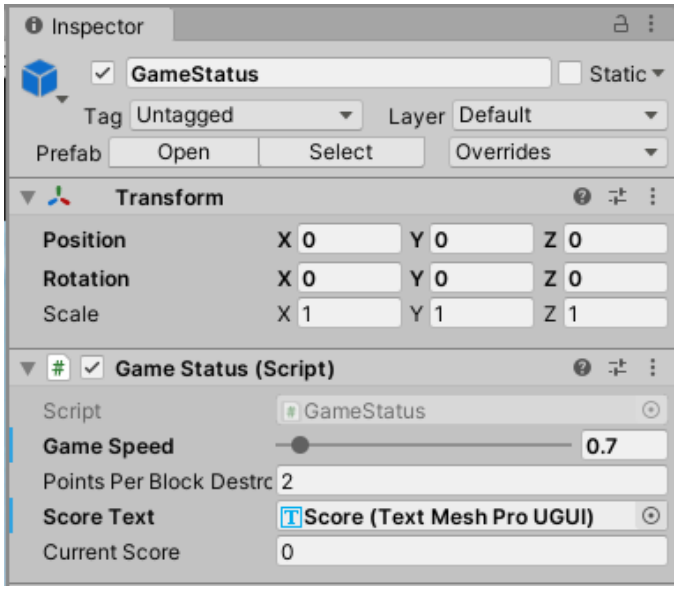
GameStatus.cs

Assembly-CSharp

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using TMPro;
5
6  public class GameStatus : MonoBehaviour
7  {
8      //config params
9      [Range(0.1f, 10f)] [SerializeField] float gameSpeed = 1f;
10     [SerializeField] int pointsPerBlockDestroyed = 2;
11     [SerializeField] TextMeshProUGUI scoreText;
12
13     //state variables
14     [SerializeField] int currentScore = 0;
15
16     // Start is called before the first frame update
17     void Start()
18     {
19         scoreText.text = currentScore.ToString();
20     }
21
22     // Update is called once per frame
23     void Update()
24     {
25         Time.timeScale = gameSpeed;
26     }
27
28     public void AddToScore()
29     {
30         currentScore += pointsPerBlockDestroyed;
31         scoreText.text = currentScore.ToString();
32     }
33 }
```

Obr. 172 Vizuál skriptu *GameStatus.cs*.

Po návrate do editora Unity je nutné inicializovať premennú *scoreText* objektom *Score* (Obr. 173).



Obr. 173 Inicializácia premennej *scoreText* v editore Unity.

Keďže objekt *GameCanvas* je *prefabom*, ktorý používame v obidvoch leveloch, je nutné všetky zmeny aplikovať na *prefab*.

Pri testovaní funkčnosti si môžeme všimnúť, že v prípade prvého levelu je skóre nastavené na 0, v prípade rozbitia bloku sa mení predpísaným spôsobom. Problém je však pri prechode z levelu do levelu, kde by sme čakali, že sa zobrazí aktuálne nahraté skóre, ktoré sa bude postupne navyšovať. Tento problém vyriešime v nasledujúcej kapitole 8.4.2.

8.4.2 Implementácia návrhového vzoru *Singleton Pattern*

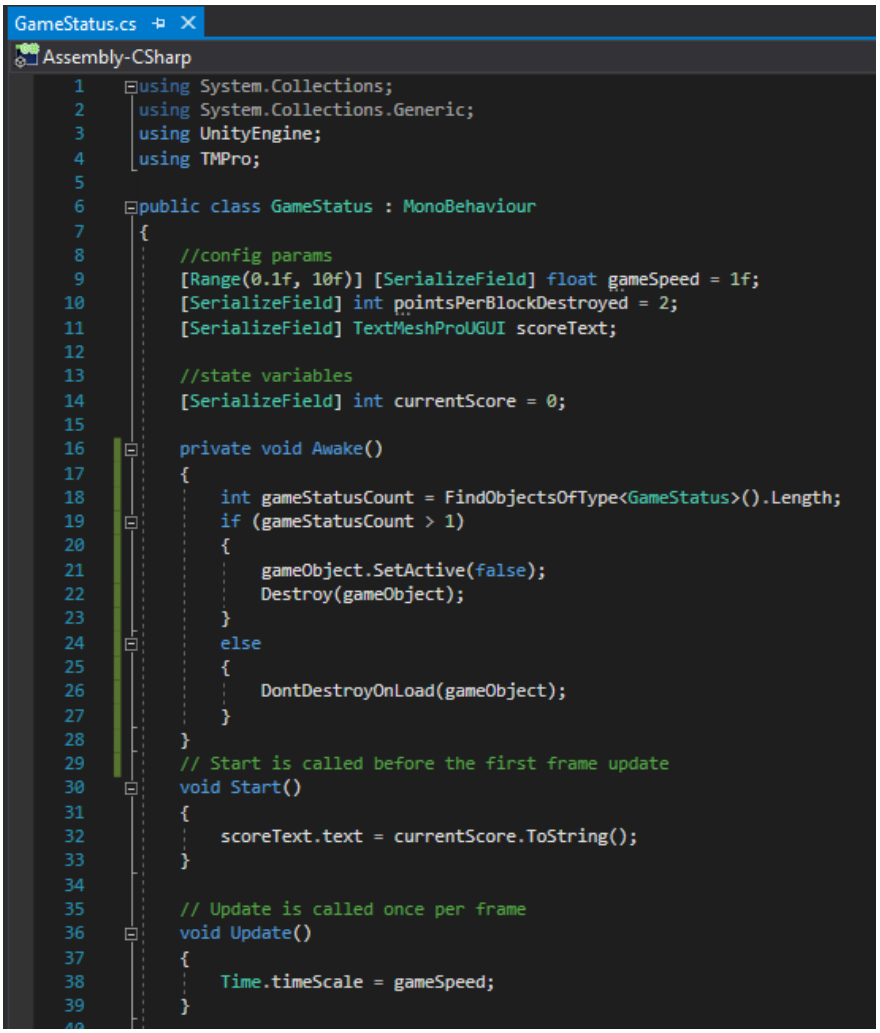
Singleton pattern je návrhový vzor (*design pattern*), ktorý používame pri programovaní, ak potrebujeme zabezpečiť, aby v celom projekte existovala len jedna inštancia danej triedy. Využitie tohto návrhového vzoru nám pomôže pri udržiavaní skóre pri prechode z jedného levelu do druhého. Problém je v tom, že v prípade prechodu na novú scénu, objekt *GameStatus*, ktorý sa stará o počítanie skóre, zaniká a všetko ako keby začalo od znovu. My ale potrebujeme aby to

fungovalo tak, aby pri načítaní ďalšej scény nedošlo k zničeniu tohto objektu, čo zabezpečíme metódou *DontDestroyOnLoad()* triedy *Object*.

Unity určuje poradie, ako aj opakované vykonanie obslužných funkcií (funkcie na obsluhu udalostí = *event functions*) počas životného cyklu (Unity, 2020o). Z toho vyplýva, že *Singleton* treba implementovať v časti *Awake*, t. j. ešte pred vykonaním funkcie *Start*().

Do skriptu *GameStatus.cs* doplníme implementáciu *Singletonu*, ktorý sa postará v prípade existencie viacerých inštancií triedy *GameStatus* o ich deaktiváciu, a v prípade existencie práve jednej inštancie, aby nedošlo k jej zničeniu. Pomocou metódy *FindObjetsOfType()*, ktorá je schopná vrátiť pole všetkých objektov špecifikovaného typu v projekte, a vďaka vlastnosti *length* triedy *Array*, spočítame všetky objekty typu *GameStatus*. V prípade existencie viacerých inštancií tieto najprv zneaktívnime a následne zničíme. Ak existuje len jedna inštancia, postaráme sa o to, aby nedošlo k jej zničeniu (Obr. 174).

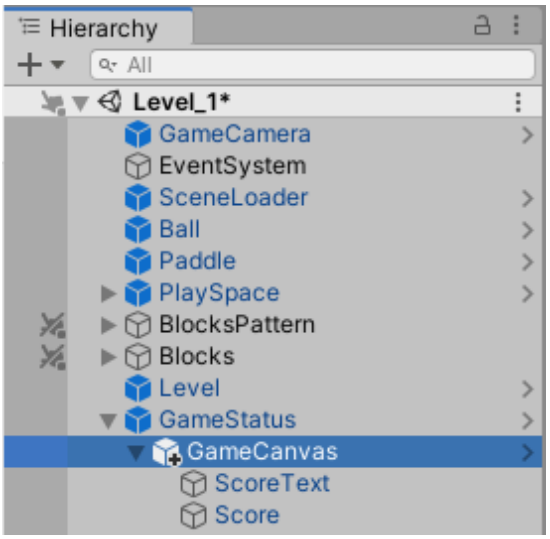
```
private void Awake()
{
    int gameStatusCount = FindObjectsOfType<GameStatus>().Length;
    if (gameStatusCount > 1)
    {
        gameObject.SetActive(false);
        Destroy(gameObject);
    }
    else
    {
        DontDestroyOnLoad(gameObject);
    }
}
```



Obr. 174 Vizuál skriptu *GameStatus.cs*.

Pozn. autora: Metóda *SetActive()* triedy *GameObject* aktivuje, resp. deaktivuje konkrétny objekt v závislosti od parametra metódy. V prípade nepoužitia tohto príkazu by sme sa mohli stretnúť s chybovým hlásením: „null reference exceptions when you're implementing the Singleton“.

Aby sa tieto zmeny správania aplikovali aj na objekt *GameCanvas*, ktorý zabezpečuje zobrazenie skóre v scéne, urobíme z neho potomka *GameStatus*, ktorý je implementovaný ako *Singleton*. *Singleton* sa aplikuje na všetkých potomkov (Obr. 175). Nezabudneme zmeny aplikovať na *prefab*. V scéne *Level_2* môžeme objekt *GameCanvas* vymazať, pretože ho tam už viac, vzhľadom na implementovaný *Singleton*, nepotrebujeme.



Obr. 175 *GameCanvas* ako potomok *GameStatus*.

Pri testovaní funkčnosti môžeme vidieť, že skóre sa pri prechode z jedného levelu do druhého modifikuje podľa očakávaní.

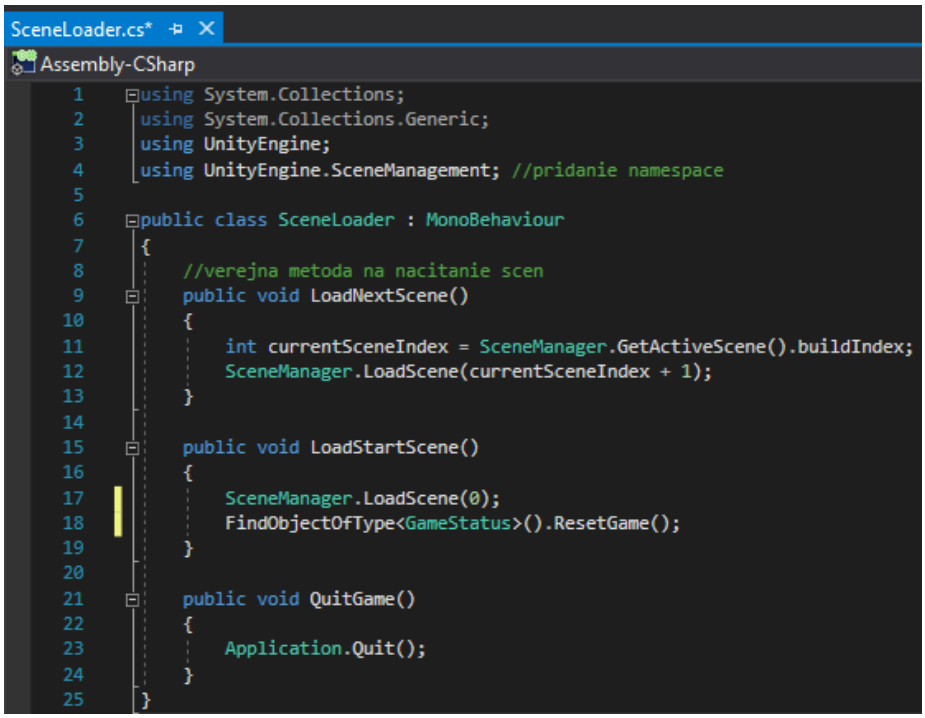
8.4.3 Nastavenie skóre na 0 v prípade začatia novej hry

Cieľom je zabezpečiť vyresetovanie skóre v prípade začatia novej hry. V tomto momente sa zobrazí aktuálne skóre, ktoré hráč získal v predošlej hre. Daný problém vyriešime vytvorením verejnej metódy *ResetGame()* v skripte *GameStatus.cs*. Táto metóda sa postará o zničenie samého seba.

```
public void ResetGame()
 {
     Destroy(gameObject);
 }
```

Dôležité je zavolať túto metódu zo správneho miesta. Tým je skript *SceneLoader.cs* a konkrétne miesto je v metóde *LoadStartScene()*, ktorá sa volá ako prvá a slúži na načítanie prvej scény hry.

```
public void LoadStartScene()
{
    SceneManager.LoadScene(0);
    FindObjectOfType<GameStatus>().ResetGame();
}
```



Obr. 176 Vizuál skriptu *SceneLoader.cs*.

Pozn. autora: V prípade ak je v skripte potrebné jedenkrát vytvoriť spojenie dvoch tried, bežne používame priamo metódu *FindObjectOfType()*. V prípade ak by ich bolo viac, je vhodné vytvoriť referenčnú premennú, ako sme to napríklad robili v skripte *Block.cs* v prípade premennej *level*. Jej použitie sa však neodporúča v rámci funkcie *Update()*, z dôvodu jej výpočtovej náročnosti.

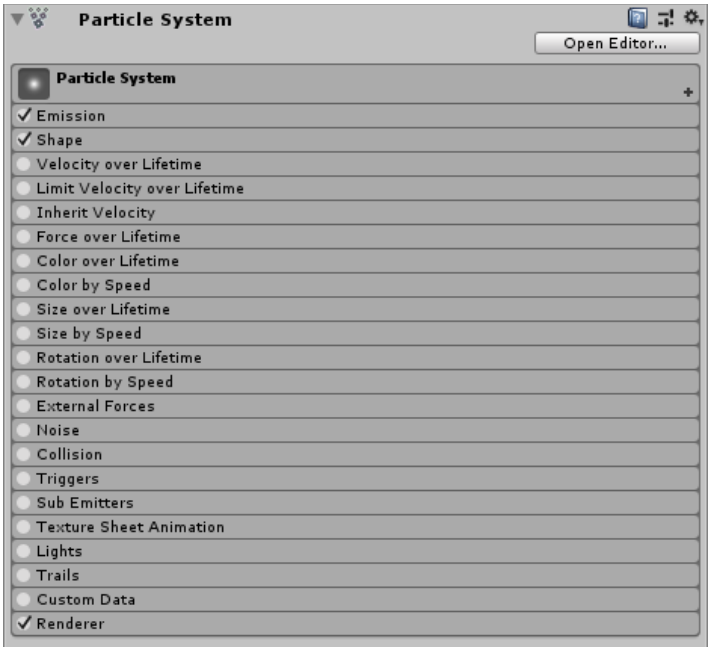
8.5 Kontrolné otázky a úlohy

1. Definujte pojmy *audio listener*, *audio source* a *audio clip*.
2. Aký komponent je nutné pridať objektu, ak chceme prehrávať zvukové záznamy?
3. Aký rozdiel je medzi metódami *Play()* a *PlayClipAtPoint()*?
4. Aký je rozdiel medzi metódami *GetComponent()* a *FindObjectOfType()*? V závislosti od čoho volíme ich použitie?
5. Objasnite implementovanú logiku počítania skóre vzhľadom na rozbité bloky.
6. Ako viete premennej priradiť rozsah, v ktorom sa môže meniť jej hodnota?

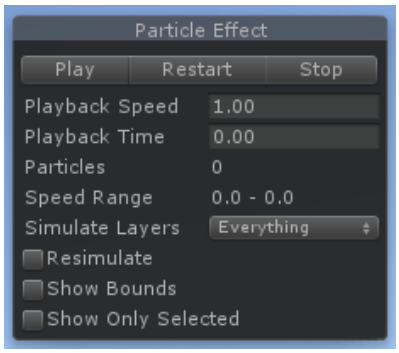
7. Ako viete zabezpečiť rôznu rýchlosť loptičky v jednotlivých leveloch?
8. Ako viete zobraziť obsah číselnej premennej v objekte typu text?
9. Ako by ste vedeli zabezpečiť, aby rôzne bloky pridávali rôznu hodnotu skóre hráčovi?
10. Aký je proces v prípade použitia vlastného fontu?
11. Definujte návrhový vzor *Singleton* a vymenujte jeho typické použitia v hrách.
12. Kedy pristupujeme k inštanciám iných tried priamo a kedy je vhodné zvoliť prístup cez referenčnú premennú?
13. Aplikujte funkcionalitu uchovávania životov a zmeňte koncept hry tak, že hra skončí až v prípade ak hráč príde o všetky životy.
14. Aplikujte prehrávanie zvukových záznamov na pozadí hry. Rozšírte danú funkcionalitu o interakciu s používateľom, aby si mohol zvoliť prehrávaný zvukový záznam, resp. vedel vypnúť všetky zvukové efekty.

9 Vytvorenie efektu – *Particle system*

Po grafickej stránke by bolo zaujímavé, ak by pri rozbití bloku nastalo prehratie nejakého vizuálneho efektu. Toto je možné zabezpečiť vytvorením *Particle system*, ktorý môžeme chápať ako komponent pridaný objektu, ktorý zabezpečuje simulovanie a renderovanie mnohých malých obrázkov nazývaných častice (*particles*), emitovaných zo zdroja (*emitter*), čo v konečnom dôsledku vytvorí dojem kompletného vizuálneho efektu. Každá jedna častica v systéme reprezentuje individuálny grafický element efektu. S výhodou sa používa pri tvorbe dynamických objektov ako je oheň, dym, kvapalina a pod., pretože je zložité takéto objekty reprezentovať modelom (*Mesh* – 3D) alebo obrázkom (*Sprite* – 2D). *Particle System* je robustný systém, ktorý disponuje mnohými vlastnosťami, ktoré sú z dôvodu prehľadnejšej práce organizované do takzvaných modulov (Obr. 177). Súčasťou systému je aj *Particle Effect panel* (Obr. 178), ktorý sa zobrazuje v okne *Scene View* a obsahuje dodatočné možnosti pre prehrávanie tvorených efektov (Unity, 2020p). Tieto skriptá nie sú primárne určené na objasnenie kompletnej práce s týmto robustným systémom, preto v prípade ak by ste nenašli čo hľadáte, odporúčame siahnuť priamo po dokumentácii Unity, kde nájdete kompletné objasnenie tejto problematiky.



Obr. 177 Moduly *Particle System*.

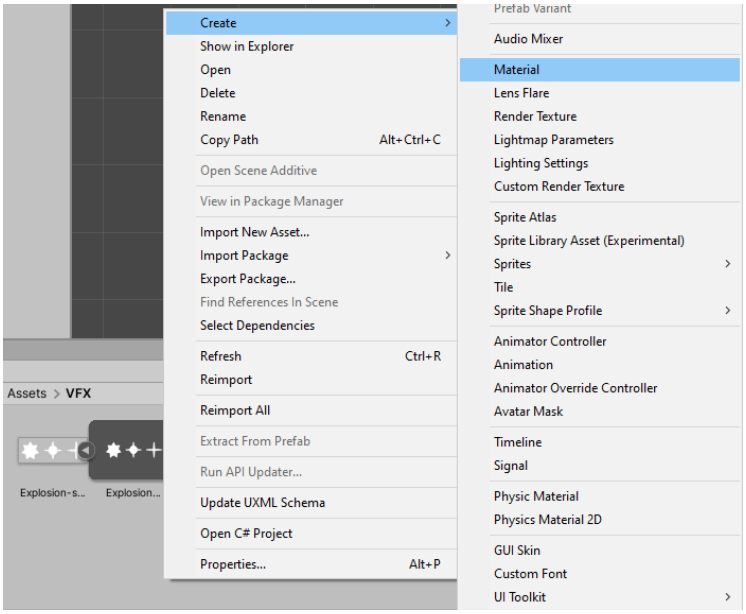


Obr. 178 *Particle Effect panel.*

Vytvoríme jednoduchý efekt explózie pomocou *spritesheet* animácie. Jednotlivé častice budú reprezentovať tri rôzne vizualizácie hviezdy, ktoré pridáme do nového priečinku s názvom VFX = *Visual effect* v našom Assets (Obr. 179).

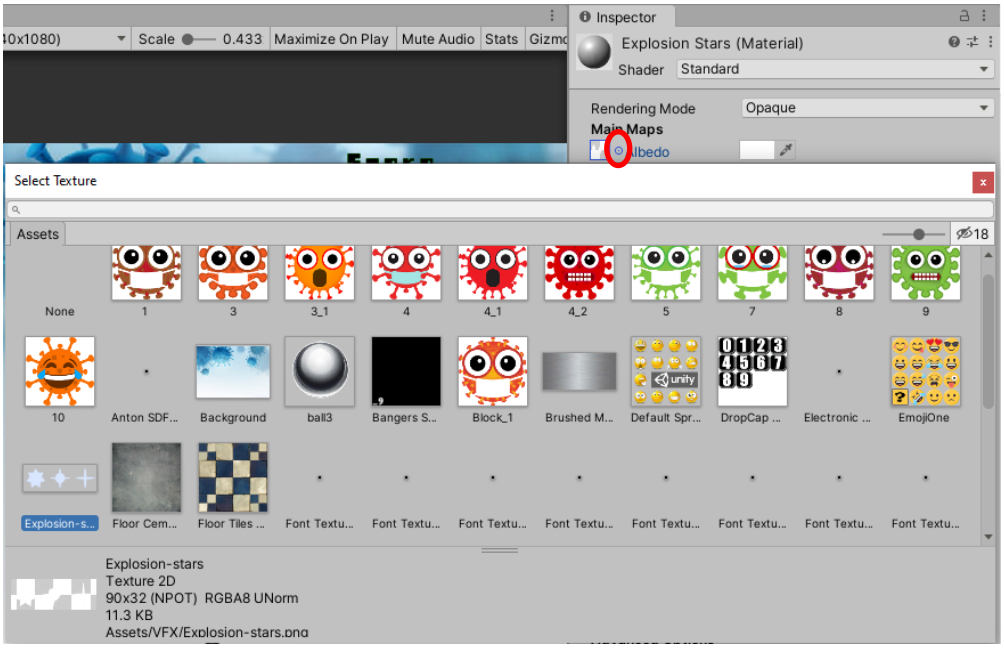
Obr. 179 Vizuál častíc pridaných do projektu.

Objektu s komponentom *particle system* nie je možné priamo priradiť daný *sprite*, ale musíme vytvoriť materiál (Obr. 180) a až ten nastaviť vo vlastnosti na to určenej. Materiál sme pomenovali *ExplosionStars*.



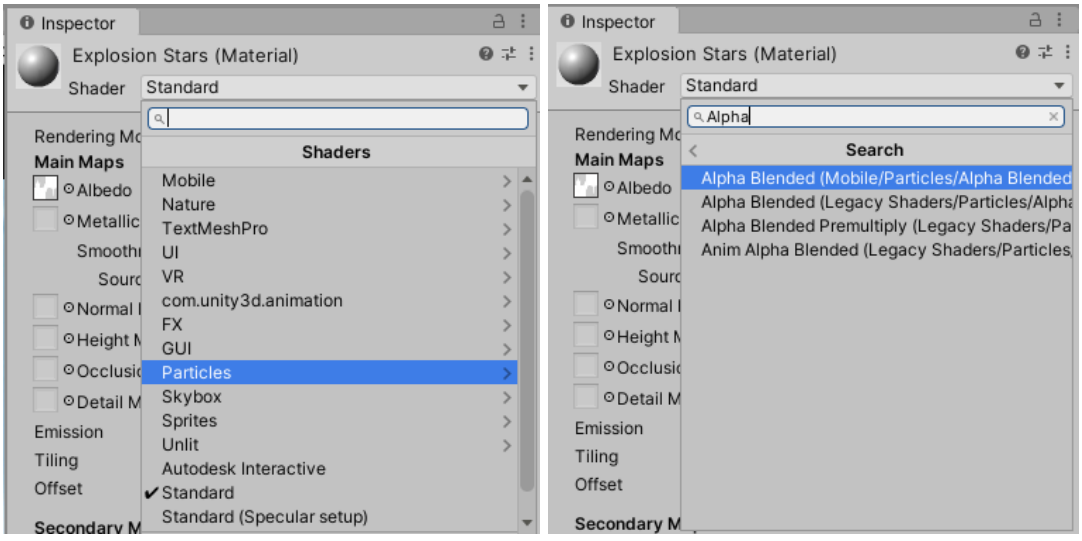
Obr. 180 Pridanie materiálu.

V okne *Inspector* vo vlastnosti *Albedo* (je potrebné kliknúť na tú bodku pred názvom *Albedo*) vyberieme *sprite* hviezd (Obr. 181).



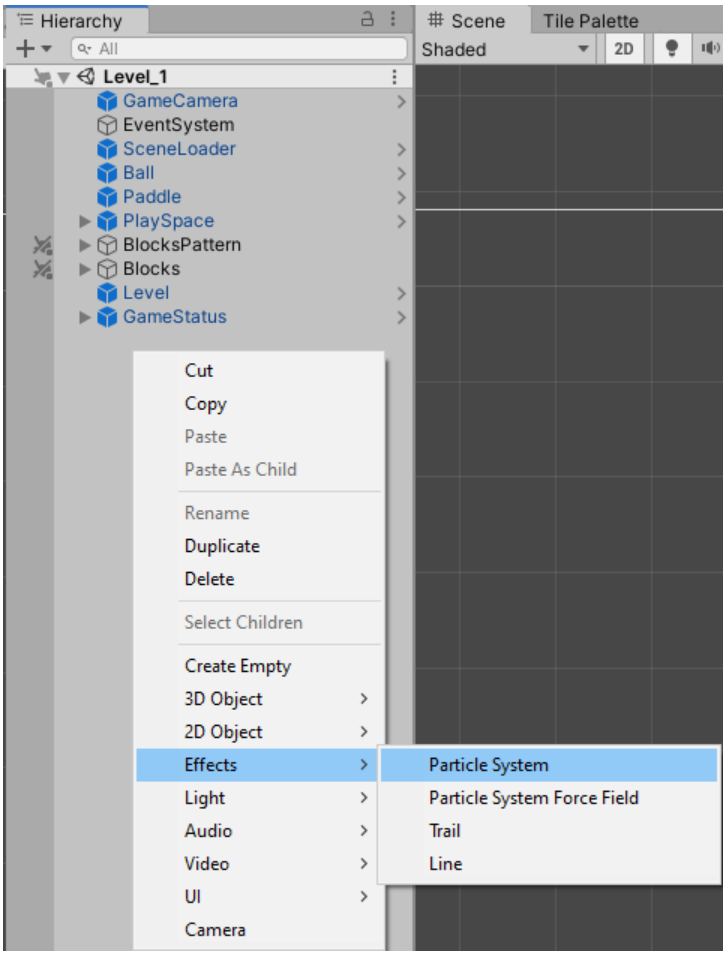
Obr. 181 Nastavenie vlastnosti *Albedo*.

Vlastnosť *Shader* nastavíme na *Particles – Alpha Blended* (Obr. 182), t. j. využívame preddefinovaný *shader*. Problematika *shaderov* je nad rámec týchto skrípt.



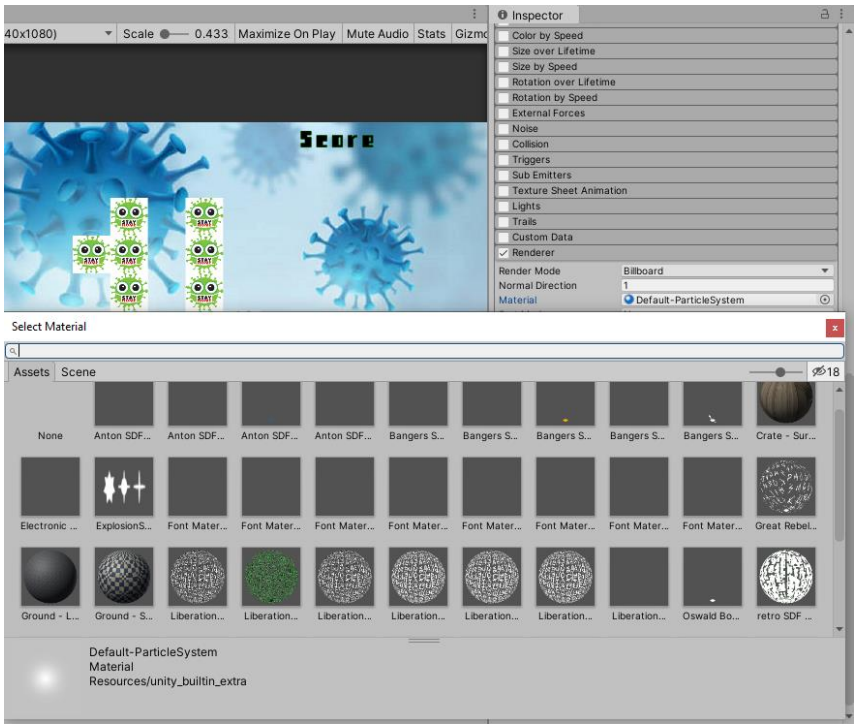
Obr. 182 Nastavenie vlastnosti *Shader*.

V okne *Hierarchy* vytvoríme nový objekt *Particle System* z kategórie *Effect* s názvom *ExplosionStars* (Obr. 183).



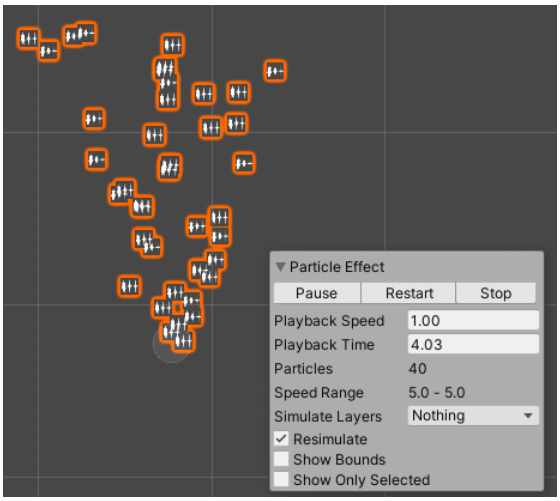
Obr. 183 Vytvorenie *Particle System*.

V okne *Inspector* nájdeme modul *Renderer* a vo vlastnosti *Material* nastavíme náš materiál *ExplosionStars* (Obr. 184).



Obr. 184 Pridanie materiálu objektu *ExplosionStars*.

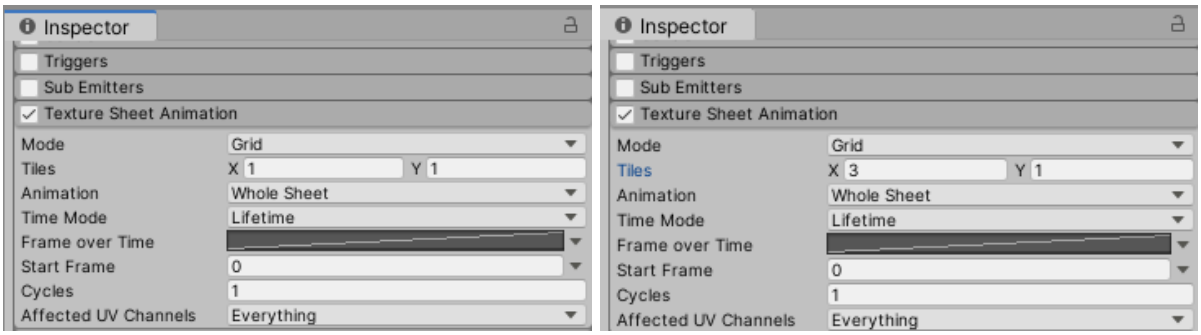
Po týchto nastaveniach môžeme vidieť pri prehrávaní tohto efektu v scéne vizuálnu reprezentáciu, ako zachytáva obrázok 185. Takáto úprava nie je našim cieľom a z toho dôvodu sa budeme venovať ďalším úpravám.



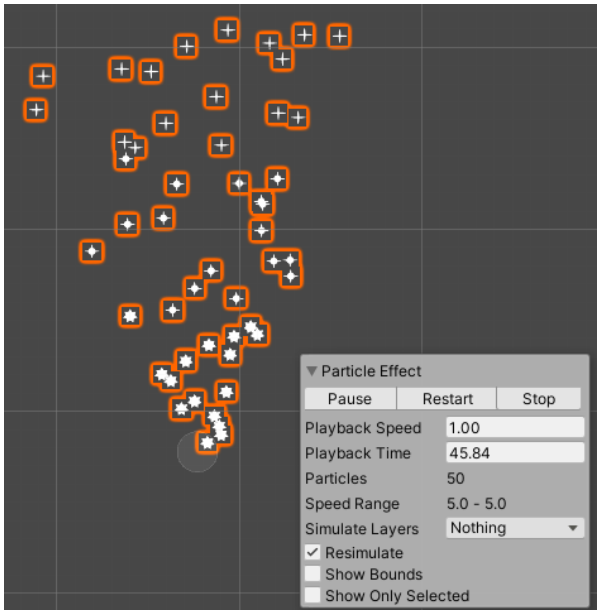
Obr. 185 Vizuál *Particle system*.

V okne *Inspector* zapneme modul *Texture Sheet Animation* (urobíme zaškrtnutím *checkboxu*), kde vlastnosť *Tiles* modifikujeme tak, že X = 3, z dôvodu že nami použitý obrázok má vlastne tri stĺpce a jeden riadok. Dôsledok tejto zmeny bude

v tom, že vždy začne systém z emitora produkovať len jeden obrázok hviezdy (Obr. 187).

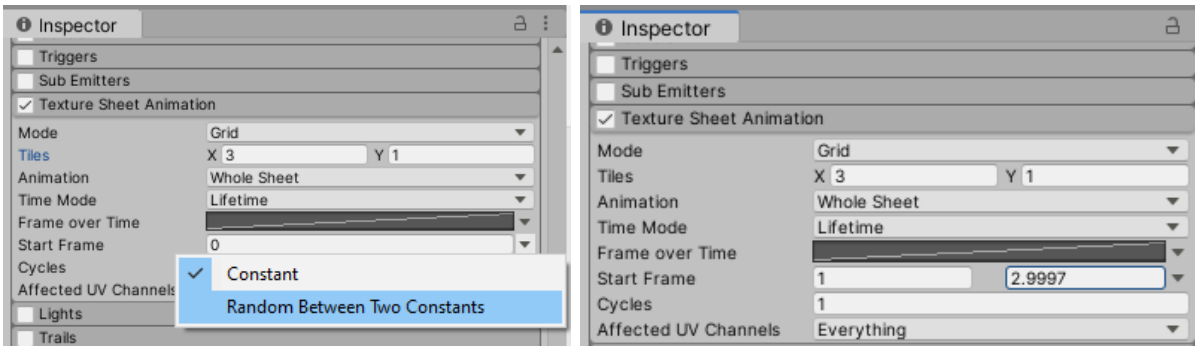


Obr. 186 Nastavenie vlastnosti *Tiles*, zobrazenie pred a po úprave.



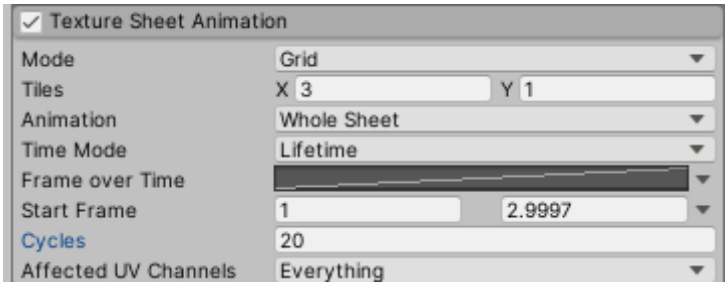
Obr. 187 Vizuál *Particle system*.

Ak chvíľu animáciu sledujeme, môžeme si všimnúť, že sa vždy začína tým istým obrázkom, a tento sa postupne mení. Aby sme do tejto simulácie výbuchu pridali trochu náhodnosti, využijeme k tomu vlastnosť *Start Frame*, kde využijeme možnosť *Random Between Two Constants* a hodnoty týchto konštánt nastavíme na 1 a 3 (Obr. 188). Výsledkom je, že vždy je emitovaná iná reprezentácia hviezdy.



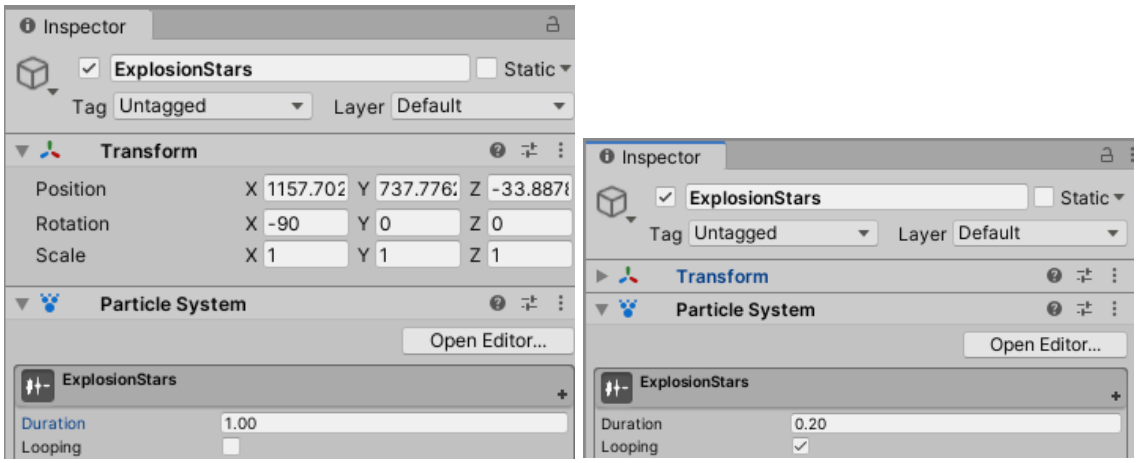
Obr. 188 Nastavenie vlastnosti *Start Frame*.

Zmenou vlastnosti *Cycles* vieme ovplyvňovať, koľkokrát sa animácia zopakuje počas životného cyklu častice. Nastavenie napr. na hodnotu 20 spôsobí blikajúci efekt, t. j. viackrát počas života tejto častice sa zmení jej tvar (Obr. 189).



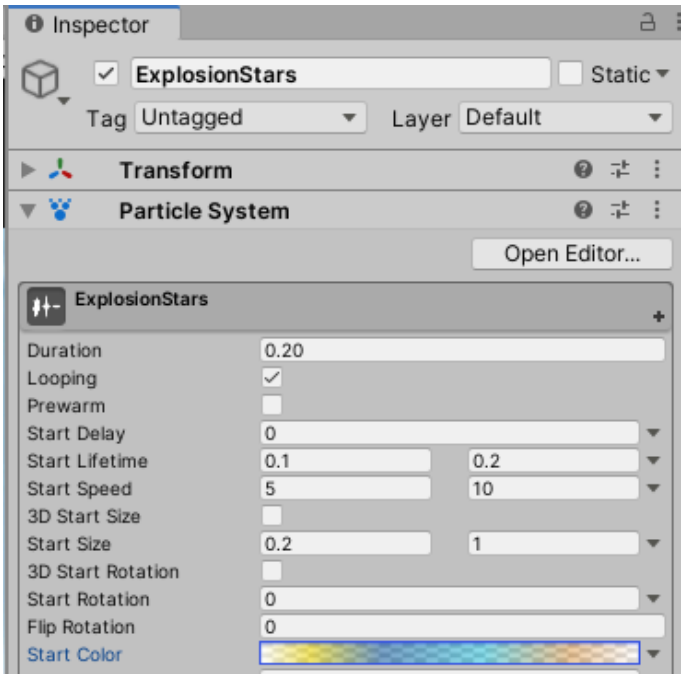
Obr. 189 Modifikácia vlastnosti *Cycles*.

Vďaka vlastnosti *Duration*, ktorá sa nachádza v základnom module (*main module*) vieme nastavovať čas trvania prehratia častice. Ideálne otestovanie tejto vlastnosti spočíva v experimentovaní zmeny týchto hodnôt pri vypnutom opakovaní (*Looping*). Vlastnosť *Duration* nastavíme na hodnotu 0,2 a *Looping* opätovne zapneme (Obr. 190).

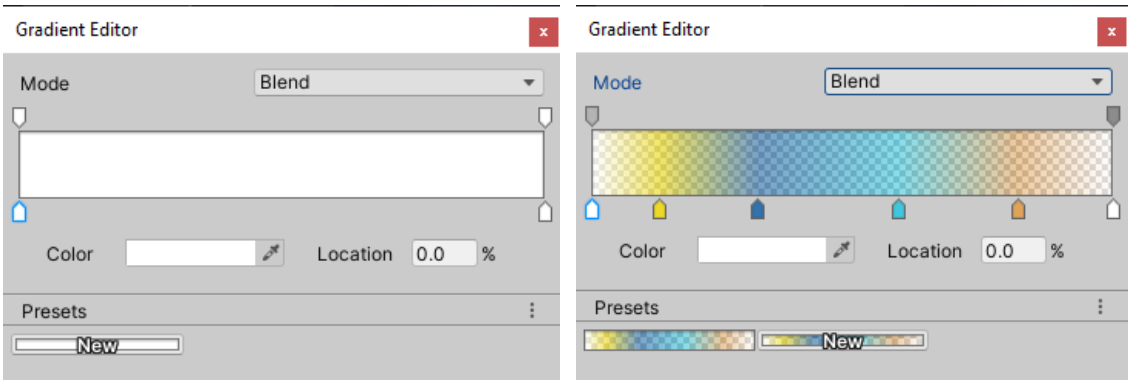


Obr. 190 Nastavenie vlastnosti *Duration*.

Pomocou vlastnosti *Start Lifetime* nastavíme trvanie existencie častice. Častica po tomto čase zaniká. Momentálne to trvá veľmi dlho, preto túto hodnotu znížime. Aby sme zabezpečili istú náhodnosť v tomto dianí, využijeme nastavenie *Random Between Two Constants*, ktoré nastavíme v rozsahu 0,1 ÷ 0,2 čo výrazne ovplyvní skrátenie života jednotlivých častíc. Taktiež zmeníme ich rýchlosť pomocou vlastnosti *Start Speed* na rozsah 5 ÷ 10. Čím je táto hodnota vyššia, tým viac sa častice rozptyľujú. Pomocou vlastnosti *Start Size* zmeníme veľkosť, ktorú nastavíme na 0,2 ÷ 1 (Obr. 191). Zmeníme aj farbu častíc pomocou vlastnosti *Start Color*, kde využijeme možnosť *Gradient*. Po výbere tejto voľby sa zobrazí *Gradient Editor*, ktorý slúži na definovanie, resp. v prípade existencie na výber gradientu. Horná časť slúži na nastavenie *Alfa* kanálu a v spodnej sa nastavuje farba (Obr. 192). Gradient si nastavte podľa vašich preferencií.

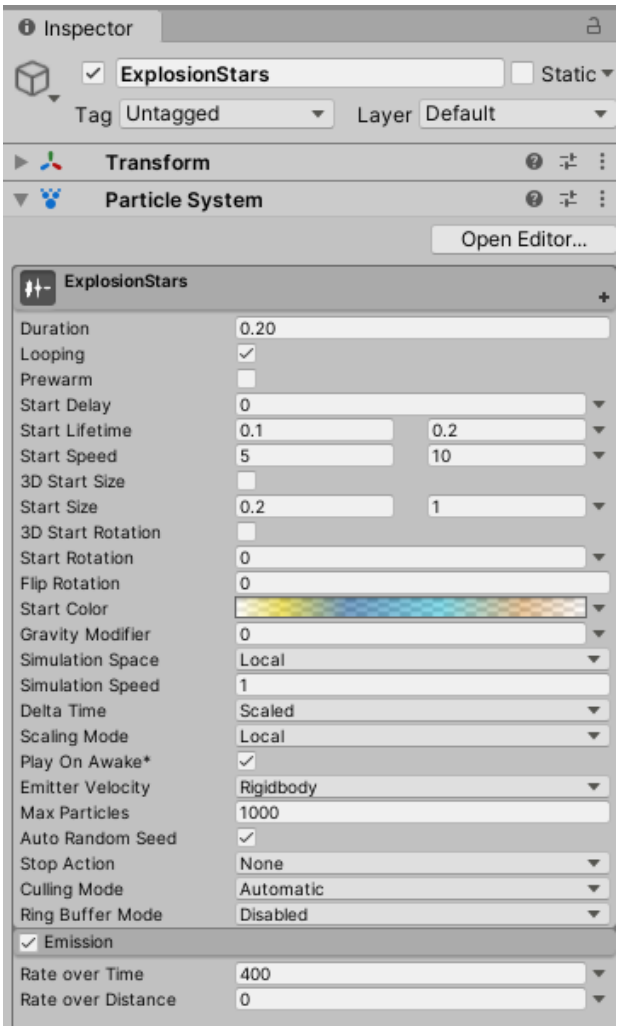


Obr. 191 Nastavenie vlastností pre *ExplosionStars*.



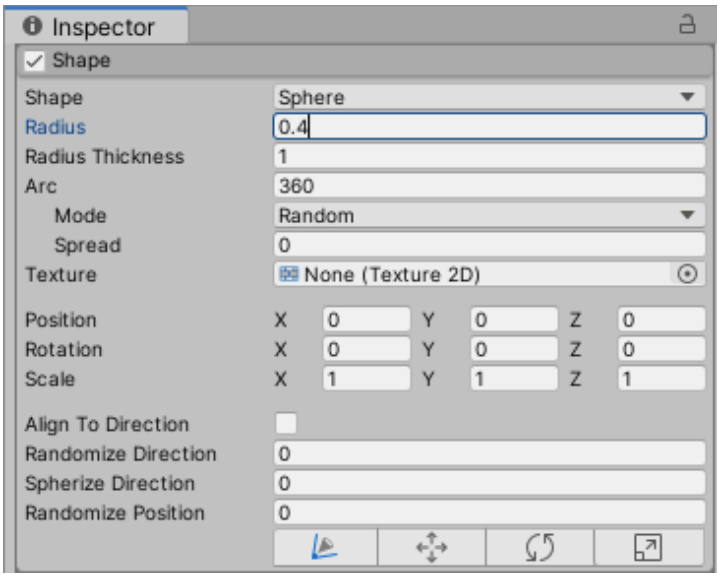
Obr. 192 Vytvorenie nového gradientu.

V module *Emission* zmeníme hodnotu vlastnosti *Rate Over Time* na 400. Táto vlastnosť určuje počet častíc emitovaných za jednotku času. Docielime tým rýchlo sa emitujúce častice v malom okolí, čo je pre simuláciu explózie vhodné.



Obr. 193 Nastavenie vlastnosti *Rate over Time*.

V module *Shape* zmeníme nastavenie vlastnosti *Shape* z *Cone* na *Sphere*, čo spôsobí, že častice budú emitované v tvare kruhu. Modifikujeme aj vlastnosť *Radius*, ktorá hovorí o veľkosti rozptylu na hodnotu 0,4 (Obr. 194).



Obr. 194 Nastavenia v module *Shape*.

Poslednú vlastnosť ktorú nastavíme je *Angular Velocity* v module *Rotation over Lifetime*, kde nastavením hodnoty v rozsahu dvoch konštánt –180 ÷ 180 dosiahneme rotovanie častíc (Obr. 195).

Obr. 195 Nastavenia v module *Rotation over Lifetime*.

Z vytvoreného *particle system* si urobíme *prefab*, ktorý umiestnime do priečinku VFX a vymažeme ho zo scény, pretože efekt bude vznikať dynamicky v momente, keď nastane rozbitie bloku.

Obr. 196 Vytvorenie *prefabu* z objektu *ExplosionStars*.

Proces inštanciácie (*instantiate*) tohto efektu doplníme do skriptu *Block.cs*. V podstate vždy keď pridáme *prefab* do scény, tak vytvárame inštanciu, alebo ak jednoducho duplikujeme objekt v okne *Hierarchy*, je to tiež proces inštanciácie.

V skripte *Block.cs* vytvoríme novú premennú typu *GameObject* s názvom *blockSparkleVFX*, ktorá bude reprezentovať náš efekt:

```
[SerializeField] GameObject blockSparkleVFX;
```

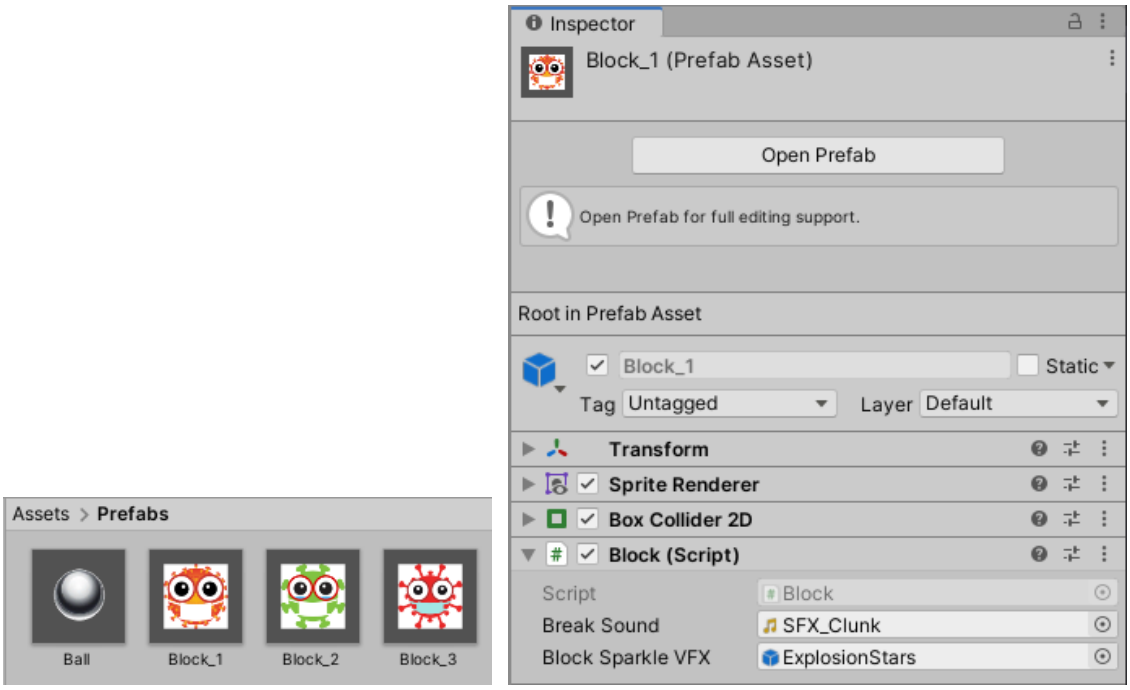
Vytvoríme novú súkromnú metódu *TriggerSparklesVFX()*, ktorá sa postará o vytvorenie inštancie počas *runtime*, t. j. počas hrania hry. Na vytvorenie inštancie je možné použiť statickú metódu *Instantiate()* triedy *Object*, ktorú je možné volať piatimi rôznymi spôsobmi v závislosti od jej deklarácie. Ak by sme nepoužili metódu, ktorá určuje aj miesto vytvorenia inštancie, tak sa inštancia vytvorí na pozícii a s rotáciou takou, ako je určené v *prefab*. My však potrebujeme aby sa vytvorila na mieste rozbitého bloku. Pomocou metódy *Destroy()* sa postaráme, aby došlo k zničeniu tohto efektu po štvrť sekunde.

```
private void TriggerSparklesVFX()
{
    GameObject sparkles =
          Instantiate(blockSparkleVFX, transform.position, transform.rotation);
    Destroy(sparkles, 0.25f);
}
```

Efekt sa má prehrať v momente, ako nastane rozbitie bloku, preto je volanie metódy potrebné doplniť do metódy *DestroyBlock()*:

```
private void DestroyBlock()
{
    AudioSource.PlayClipAtPoint(breakSound, Camera.main.transform.position);
    Destroy(gameObject);
    level.BlockDestroyed();
    FindObjectOfType<GameStatus>().AddToScore();
    TriggerSparklesVFX();
    //Debug.Log(collision.gameObject.name);
}
```

Po zmenách v skripte *Block.cs* a návrate do editora Unity je nutné inicializovať premennú *blockSparkleVFX* pre každý blok pri ktorom sa má tento efekt uplatniť. Bloky, ktoré je možné rozbiť, sú tri: *Block_1*, *Block_2* a *Block_3*. Zmeny realizujeme priamo na *prefabe* týchto blokov, aby sa aplikovali na všetky inštancie použité v scénach hry.



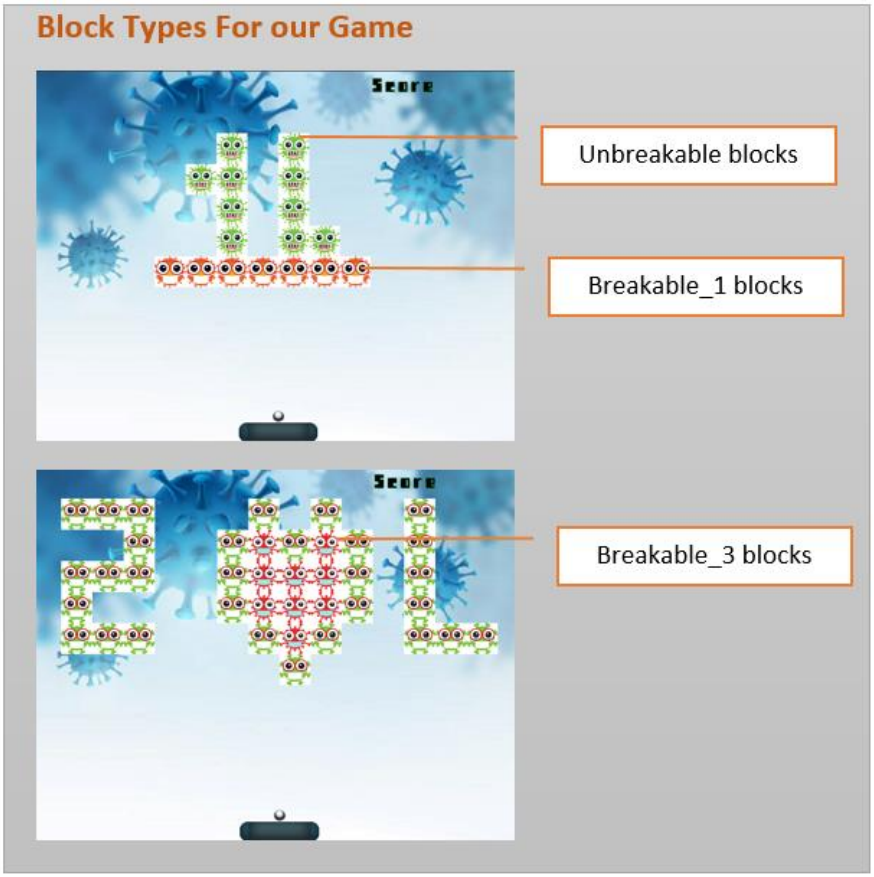
Obr. 197 Ukážka inicializácia premennej *blockSparkleVFX* jednotlivých *prefabov*.

9.1 Kontrolné otázky a úlohy

1. Ako je možné v Unity vytvoriť nejaký vizuálny efekt?
2. Definujte pojmy *Particle System* a *Particle Effect panel*.
3. Akými modulmi disponuje *Particle System*?
4. Pomocou akej metódy vieme vytvoriť inštanciu v kóde?
5. Vytvorte si vlastný efekt výbuchu a experimentujte s jednotlivými nastaveniami vlastností v moduloch *Particle System*.
6. Vytvorte vlastný efekt, ktorý sa prehrá napríklad pri výhre, resp. prehre hráča, pri postupe do ďalšieho levelu a pod.

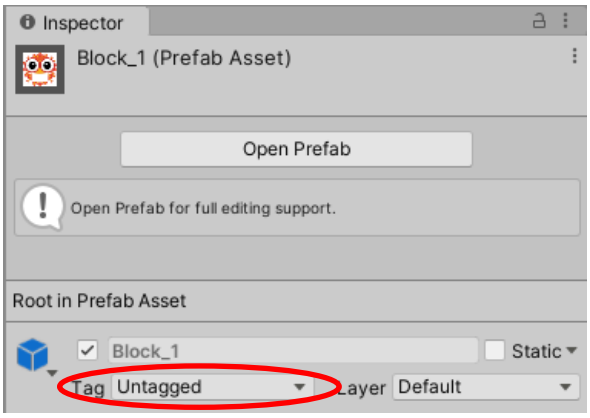
10 Diferenciácia blokov použitím *tagov*

V hre využijeme *tagy* na určenie blokov, ktoré sú nerozbitné (*unbreakable blocks*), rozbitné na jeden úder (*breakable_1 blocks*) a bloky pri ktorých je nutné rozbiť ich trikrát (*breakable_3 blocks*) (Obr. 198). *Tagy* vo všeobecnosti uľahčujú prácu pri programovaní, konkrétne pri identifikácii objektov.



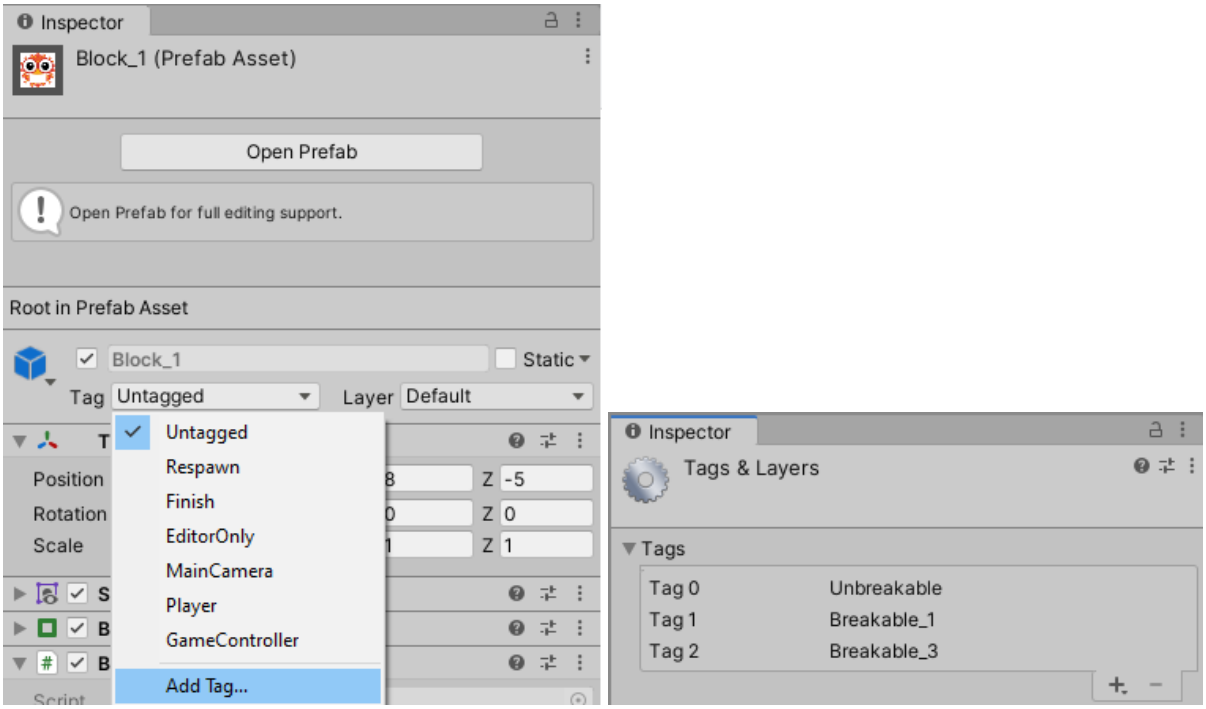
Obr. 198 Rôzne typy blokov.

V okne *Inspector prefabu Block_1* môžeme vidieť že nemá priradený žiadny *tag* (Obr. 199).



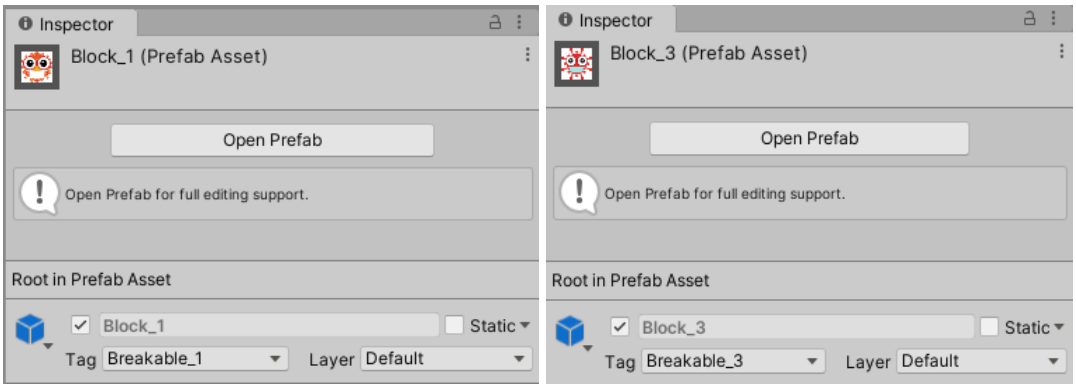
Obr. 199 *Block_1* bez *tagu*.

Vytvoríme si tri rôzne *tagy* a to s pomocou rozbalenia tejto ponuky a možnosti *Add Tag*. Pomocou tlačidla „+“ pridáme tri rôzne *tagy*, ako je ilustrované obrázkom 200.



Obr. 200 Vytvorenie *tagov*.

Prefabom *Block_1* a *Block_2* nastavíme *tag Breakable_1*, prefabu *Block_3* nastavíme *tag Breakable_3* a prefabu *Unbreakable_block tag Unbreakable* (Obr. 201).



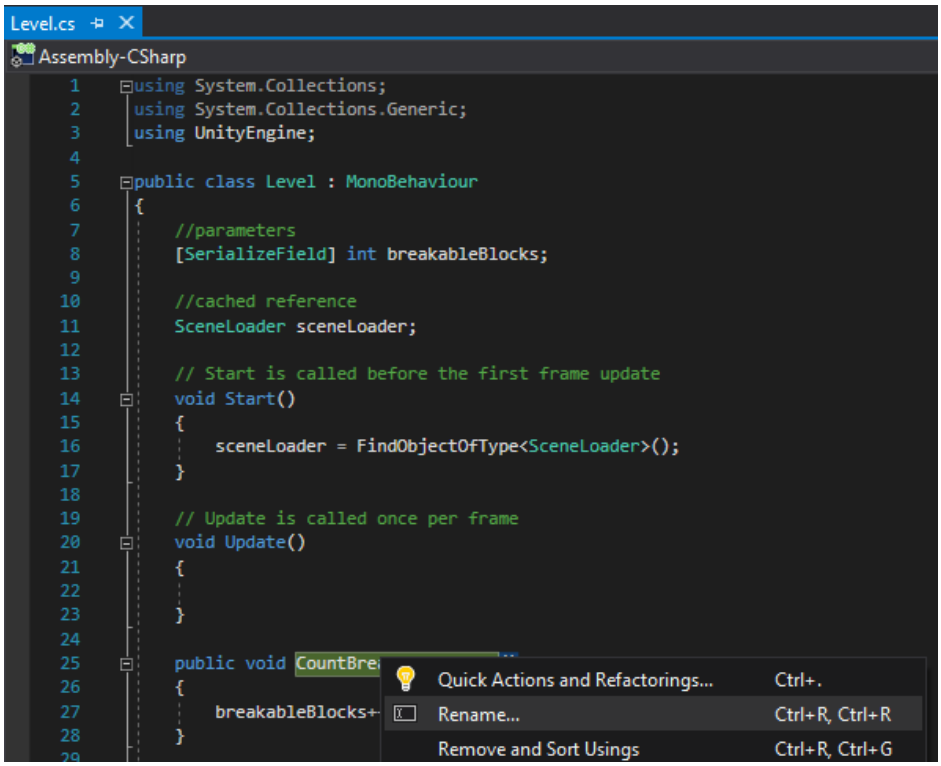
Obr. 201 Ilustrácia priradenia *tagov prefabom*.

V skripte *Block.cs* upravíme volanie metódy *DestroyBlock()* len v prípade *tagov Breakable_1* a *Breakable_3*.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if(tag == "Breakable_1" || tag == "Breakable_3")
    {
        DestroyBlock();
    }
}
```

V skripte *Level.cs* máme vytvorenú metódu *CountBreakableBlocks()*, ktorá počíta koľko blokov bolo rozbitých. Funkcionalitu tejto metódy nemeníme, modifikujeme však jej názov na *CountBlocks()*, ktorý presnejšie vystihuje jej podstatu.

Pozn. autora: Využívame na to kontextovú ponuku, aby sa metóda premenovala v celom projekte, t. j. všade kde treba (Obr. 202).



Obr. 202 Využitie kontextovej ponuky na premenovanie existujúcej metódy.

Následne sa vrátime k úprave funkcie *Start()* v skripte *Block.cs*, kde zavolanie tejto metódy podmienime len v prípade vyhovujúcich *tagov*:

```
private void Start()
{
    level = FindObjectOfType<Level>();
    if (tag == "Breakable_1" || tag == "Breakable_3")
    {
        level.CountBlocks();
    }
}
```

Z dôvodu prehľadnosti je možné túto časť kódu zapuzdriť s pomocou kontextovej ponuky a možnosti *Quick Actions and Refactoring* do novej metódy s názvom *CountBreakableBlocks()*:

```
private void Start()
{
    CountBreakableBlocks();
}
private void CountBreakableBlocks()
{
    level = FindObjectOfType<Level>();
    if (tag == "Breakable_1" || tag == "Breakable_3")
    {
        level.CountBlocks();
    }
}
```

Pri testovaní funkčnosti je možné si všimnúť, že bloky s *tagom Unbreakable* sa nezapočítavajú do počtu blokov, ktoré je potrebné rozbiť – čo reprezentuje premenná *breakableBlocks* deklarovaná v skripte *Level.cs*. A tiež, že tieto bloky pri strete s loptičkou ostávajú.

10.1 Zabezpečenie funkcionality pre blok, ktorý je nutné trafiť trikrát

Ak chceme mať v hre bloky, ktoré je nutné trafiť viackrát aby boli zničené, musíme túto funkcionalitu naimplementovať. V skripte *Block.cs* deklarujeme dve nové premenné. Premenná *maxHits* určuje koľkokrát je nutné blok trafiť, aby bol zničený a premenná *timesHit* je počítadlo, ktoré počíta koľkokrát už bol blok zasiahnutý.

```
//configuration params
[SerializeField] int maxHits;

//state variables
int timesHit;
```

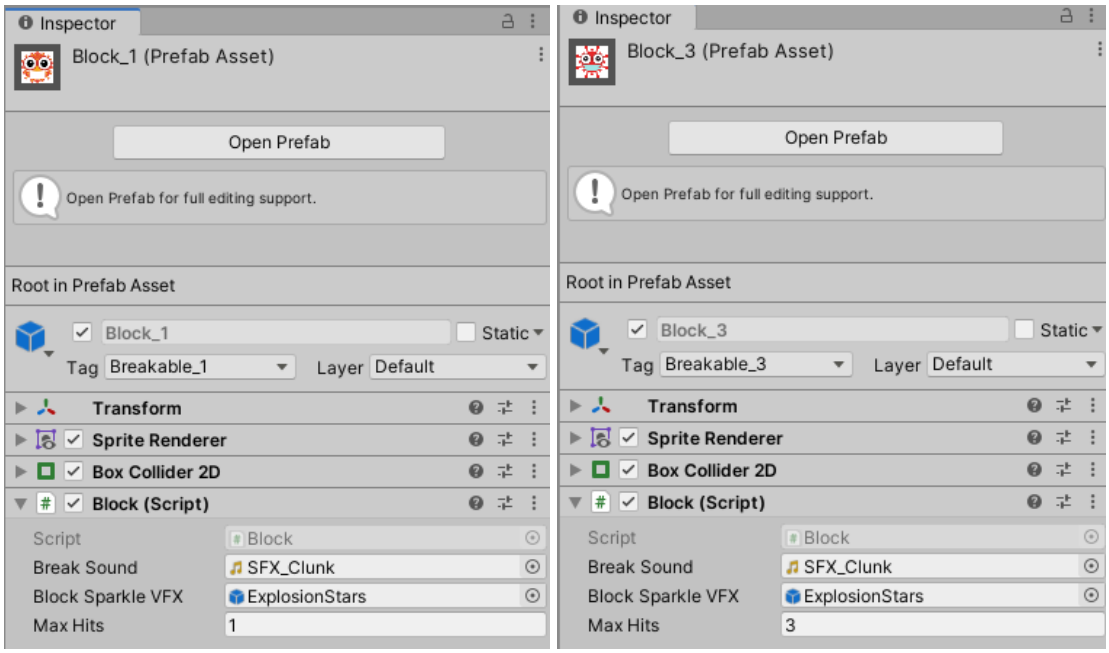
V prípade ak dôjde ku kolízii loptičky s blokom je potrebné premennú *timesHit* inkrementovať:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if(tag == "Breakable_1" || tag == "Breakable_3")
    {
        timesHit++;
        DestroyBlock();
    }
}
```

K zničeniu bloku však má prísť len v prípade, ak počet zasiahnutí je totožný s premennou *maxHits*. V tomto prípade sa zvykne pre istotu použiť relačný operátor >=, ako len == z dôvodu, že skôr ako sa daná podmienka vyhodnotí, mohlo by prísť k ďalšej kolízii a k navýšeniu premennej *timesHit*.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if(tag == "Breakable_1" || tag == "Breakable_3")
    {
        timesHit++;
        if(timesHit >= maxHits)
        {
            DestroyBlock();
        }
    }
}
```

Po týchto úpravách je nutné inicializovať premenné pre jednotlivé bloky v editore Unity. Zmeny realizujeme priamo na *prefaboch*. Pre *Block_1* a *Block_2* nastavíme hodnotu premennej *maxHits* na 1. Prefabu *Block_3* nastavíme hodnotu tejto premennej na 3.



Obr. 203 Nastavenie premennej *maxHits*.

Otestovaním funkčnosti môžeme vidieť, že k zničeniu jednotlivých blokov dochádza podľa nastavení premennej *maxHits*.

10.1.1 Zmena vzhľadu bloku v závislosti od hodnoty premennej *maxHits*

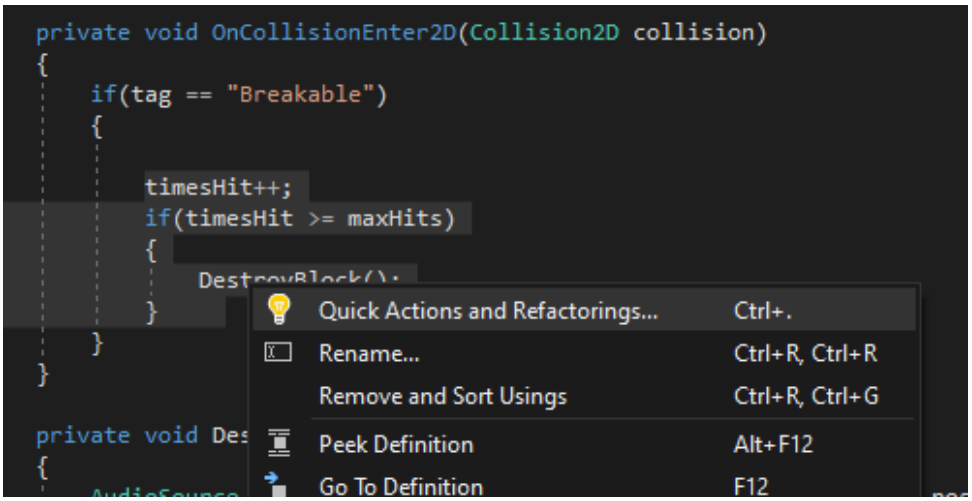
Cieľom je v závislosti od nastavení hodnoty v premennej *maxHits* modifikovať *sprity* daného bloku, t. j. po každom zásahu zmeniť jeho vzhľad, aby hráč vedel čo sa deje. V tomto kontexte sa môžeme stretnúť s pojmom „*affordance*“, ktorý definoval Gibson (Greeno, 1994). *Affordance* vlastne nejakým spôsobom reprezentuje čo sa deje, napríklad, v našom prípade zmenou vizuálu bloku pri jednotlivých rozbitiach hráč pochopí, že ide o blok, ktorý treba zničiť viacerými údermi.

V skripte *Blocks.cs* si vytvoríme pole *spritov*, do ktorých uložíme *sprity* reprezentujúce rôzny vizuál bloku tak, ako ilustruje obrázok 204.

Obr. 204 Vizuál pre *Block_3*.

```
[SerializeField] Sprite[] hitSprites;
```

Časť kódu ako ilustruje obrázok 205 zapuzdrime do novej metódy s názvom *HandleHit()* a následne rozšírime túto funkcionalitu.



Obr. 205 Vytvorenie metódy *HandleHit()*.

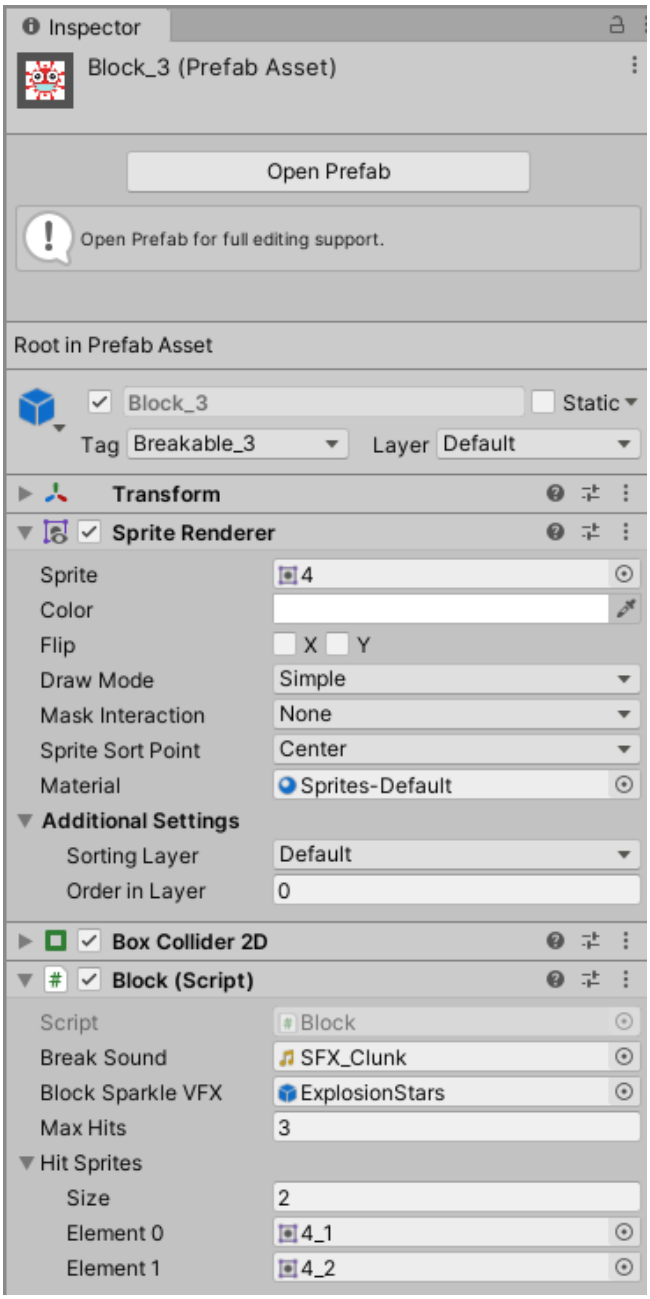
Vytvoríme novú metódu *ShowNextHitSprite()*, ktorá zabezpečí zobrazenie požadovaného *spritu* v momente, keď má dôjsť ku zmene, t. j. keď bol blok zasiahnutý. Logika tejto funkcionality je postavená na zmene *spritu*, ktorý je uložený v deklarovanom poli. Prvý element poľa má vždy index rovný 0, z toho dôvodu modifikujeme hodnotu premennej *timesHit* o 1. Vlastnosť *sprite* komponentu *SpriteRenderer* hovorí o tom, ktorý *sprite* sa má zobraziť.

```
private void ShowNextHitSprite()
{
    int spriteIndex = timesHit - 1;
    GetComponent<SpriteRenderer>().sprite = hitSprites[spriteIndex];
}
```

Túto metódu je potrebné zavolať vždy keď je blok zasiahnutý. Preto modifikujeme aj telo metódy *HandleHit()*.

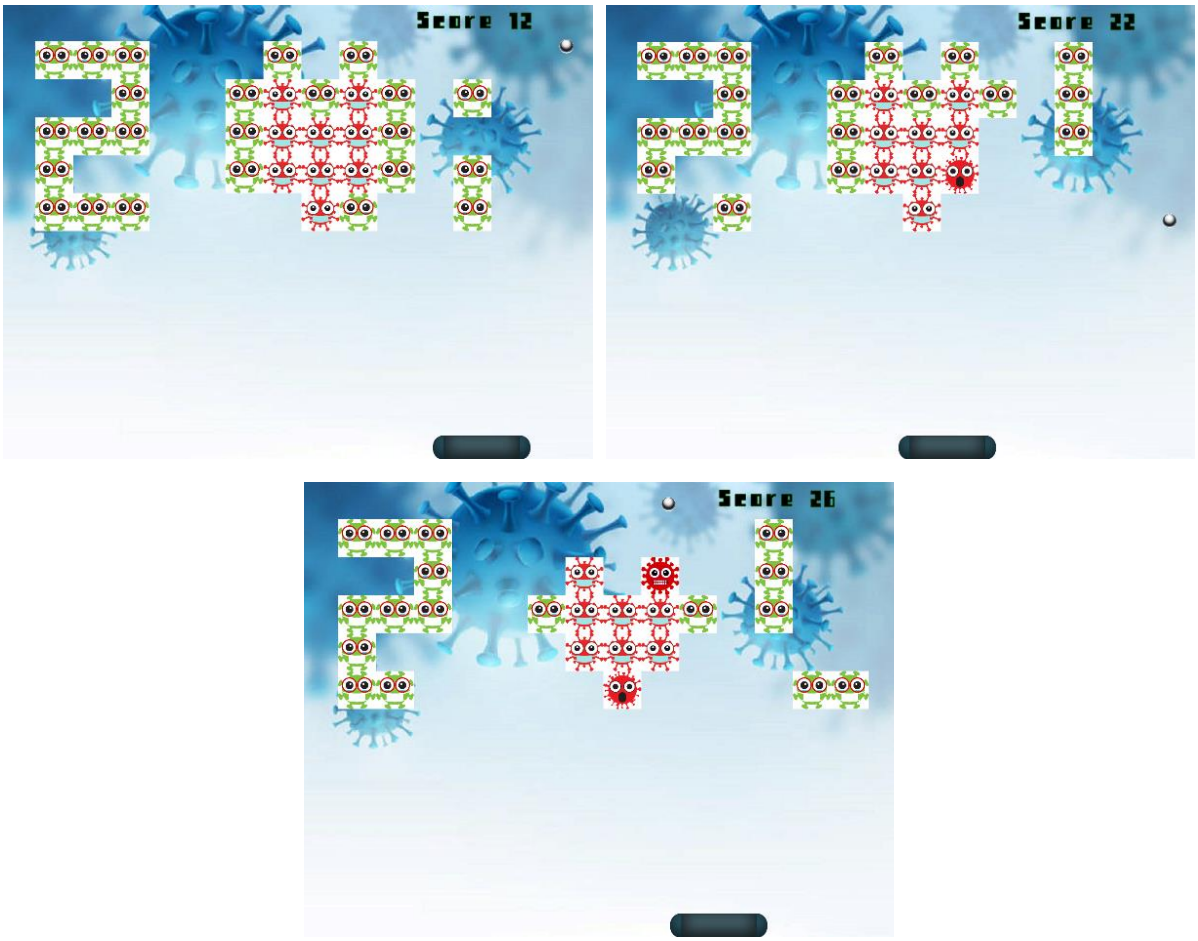
```
private void HandleHit()
{
    timesHit++;
    if (timesHit >= maxHits)
    {
        DestroyBlock();
    }
    else
    {
        ShowNextHitSprite();
    }
}
```

Po návrate do editora Unity je nutné na *prefabe Block_3* inicializovať deklarované pole *hitSprites*. Jeho veľkosť nastavíme na hodnotu 2 a inicializujeme jednotlivé elementy poľa konkrétnymi grafickými reprezentáciami (Obr. 206). Pole inicializujeme hodnotou dva, pretože pôvodná podoba bloku je nastavená pomocou vlastnosti *Sprite*, t. j. pole obsahuje len vizualizácie bloku, ktoré sa budú meniť vzhľadom na zásah bloku.



Obr. 206 Nastavenie vlastností pre *prefab Block_3*.

Výsledkom tejto funkcionality bude, že sa vizuál objektu *Block_3* bude pri jednotlivých úderoch meniť až kým nepríde k jeho zničeniu (Obr. 207).



Obr. 207 Vizuál zachytávajúci zmenu vizuálu objektu *Block_3*.

10.2 Kontrolné otázky a úlohy

1. Definujte čo sú *tagy* a v akých iných situáciách by bolo možné ich použiť.
2. Objasnite logiku zabezpečenia funkcionality, že niektoré bloky je potrebné trafiť viackrát aby nastalo ich zničenie.
3. Experimentujte s implementovanou funkcionalitou a vytvorte bloky, ktoré je nutné trafiť napríklad dvakrát, štyrikrát a pod.
4. Vytvorte si vlastný grafický aparát, ktorý bude reprezentovať rozbitie bloku po jednotlivých úderoch.

11 Finálne úpravy hry

Aby sme mohli pohodlne otestovať hru vytvoríme si *autoplayera*, t. j. mechanizmus, ktorý bude hrať hru za nás a my sa môžeme plne venovať sledovaniu testovaných funkcionalít. Túto funkcionalitu doplníme do skriptu *GameStatus.cs*, kde vytvoríme novú logickú premennú *isAutoPlayEnabled* podľa ktorej budeme definovať, či je *autoplayer* zapnutý, alebo nie.

```
[SerializeField] bool isAutoPlayEnabled;
```

Definujeme aj verejnú metódu *IsAutoPlayEnabled()*, ktorá nám vráti hodnotu tejto premennej:

```
public bool IsAutoPlayEnabled()
{
    return isAutoPlayEnabled;
}
```

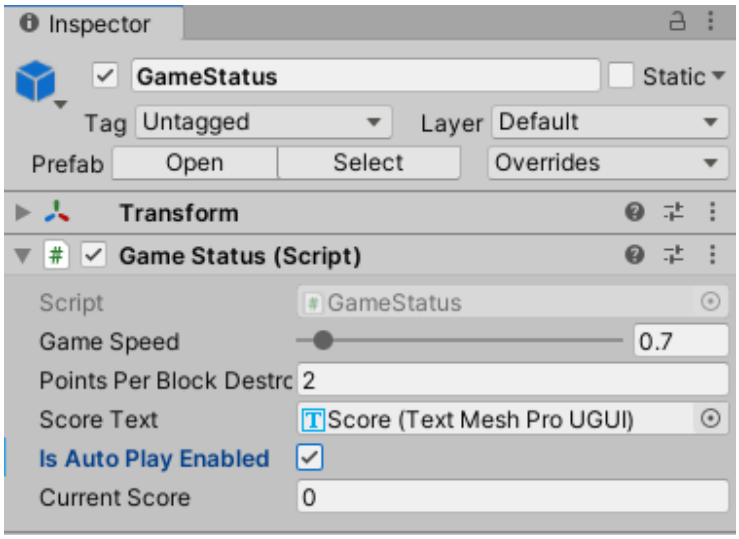
V skripte *Paddle.cs* vytvoríme novú súkromnú metódu *GetXPos()*, ktorá bude kontrolovať či je *autoplayer* zapnutý alebo vypnutý. V prípade jeho zapnutia nastaví pozíciu padla na základe pozície objektu *Ball*. V opačnom prípade necháme nastavenie pozície padla tak ako bolo, t. j. v strede obrazovky.

```
private float GetXPos()
{
    if(FindObjectOfType<GameStatus>().IsAutoPlayEnabled())
    {
        return FindObjectOfType<Ball>().transform.position.x;
    }
    else
    {
        return Input.mousePosition.x / Screen.width * screenWidthInUnits;
    }
}
```

Modifikácie robíme aj vo funkcii *Update()*, kde x-ovú súradnicu padla nastavíme podľa návratovej hodnoty metódy *GetXPos()*.

```
void Update()
{
    Vector2 paddlePos = new Vector2(transform.position.x, transform.position.y);
    paddlePos.x = Mathf.Clamp(GetXPos(), minX, maxX);
    transform.position = paddlePos;
}
```

Zaškrtnutím *checkboxu* pri premennej *isAutoPlayEnabled* v editore Unity môžeme zapínať a vypínať implementované automatizované testovanie (Obr. 208).



Obr. 208 Zapnutie a vypnutie autoplayera.

Metóda *FindObjectOfType()* je tak výpočtovo zložitá, že by sme ju nemali volať v *Update()* funkcii, ktorá sa volá každý *frame*. Je preto vhodnejšie deklarovať v skripte *Paddle.cs* referenčné premenné a inicializovať ich vo funkcii *Start()*.

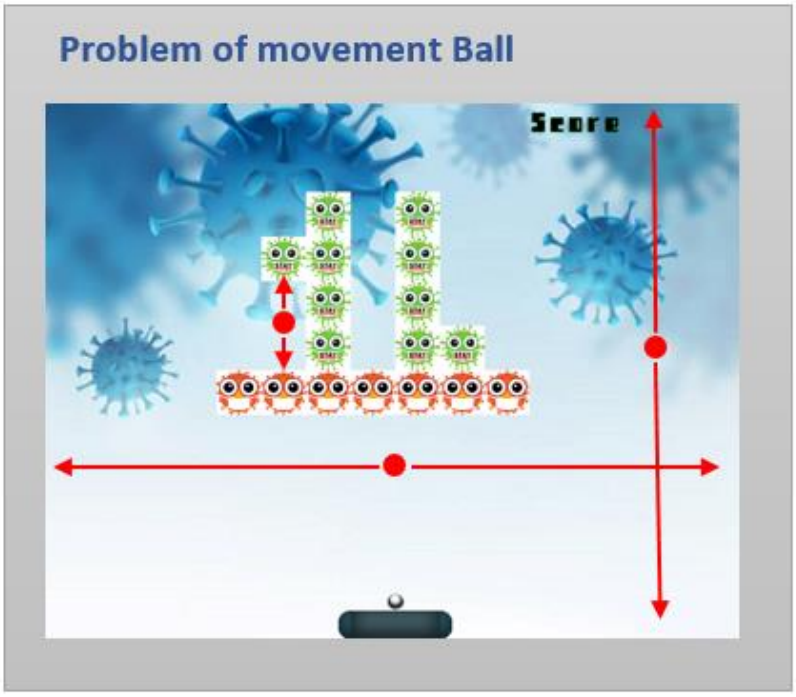
```
//cached references
GameStatus theGameStatus;
Ball theBall;

void Start()
{
    theGameStatus = FindObjectOfType<GameStatus>();
    theBall = FindObjectOfType<Ball>();
}
```

Vzhľadom na doplnené deklarácie premenných je nutné upraviť aj metódu *GetXPos()*:

```
private float GetXPos()
{
    if(theGameStatus.IsAutoPlayEnabled())
    {
        return theBall.transform.position.x;
    }
    else
    {
        return Input.mousePosition.x / Screen.width * screenWidthInUnits;
    }
}
```

11.1 Zabezpečenie nezacyklenia sa loptičky

Pri testovaní sme si mohli všimnúť, že sa občas môžu vyskytnúť momenty, keď loptička ako keby uviazla v slučke, t. j. odráža sa v horizontálnom resp. vertikálnom smere medzi dvomi objektmi akoby bez posuvu (Obr. 209).



Obr. 209 Možné problémy pri pohybe loptičky.

V skripte *Ball.cs* definujeme premennú *randomFactor*, ktorá bude slúžiť na zmenu odrazu loptičky. Inicializujeme ju na hodnotu 1:

```
[SerializeField] float randomFactor = 1f;
```

Zavedieme aj referenčné premenné *myRigidBody2D* a *myAudioSource*, ktoré inicializujeme vo funkcii *Start()*.

```
//cached component references
Rigidbody2D myRigidBody2D;
AudioSource myAudioSource;


void Start()
{
    paddleToBallVector = transform.position - paddle1.transform.position;
    myRigidBody2D = GetComponent<Rigidbody2D>();
    myAudioSource = GetComponent<AudioSource>();
}
```

Z dôvodu týchto úprav je nutné upraviť metódu *LaunchOnMouseClick()*, ako aj *OnCollisionEnter2D()*, ktoré voláme vo funkcii *Update()* z dôvodu ako sme vysvetlili vyššie.

```
private void LaunchOnMouseClick()
{
    if (Input.GetMouseButtonDown(0))
    {
        myRigidBody2D.velocity = new Vector2(xPush, yPush);
        hasStarted = true;
    }
}
```

V *OnCollisionEnter2D()* využívame metódu *Range()* triedy *Random*, čím zabezpečíme náhodné generovanie hodnoty v danom rozsahu a tým predídeme zacykleniu loptičky.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    Vector2 velocityTweak =
          new Vector2(UnityEngine.Random.Range(0f, randomFactor),
                      UnityEngine.Random.Range(0f, randomFactor));
    if (hasStarted)
    {
        myAudioSource.Play();
        myRigidBody2D.velocity += velocityTweak;
    }
}
```

Pozn. autora: V prípade, ak by sme hodnotu premennej *randomFactor* nastavili na vyššiu hodnotu, napr. 10, tak by bolo pozorovateľné možné zacyklenie pohybu loptičky, ako sme objasnili vyššie.

Iný spôsob ako predísť tomuto nežiadúcemu efektu zacyklenia je, obkolesiť hracie prostredie kruhom, a nie hranatými stenami ako teraz. Tým by sa loptička vždy odrazila pod iným uhľom a nemuseli by sme tento problém riešiť pomocou vlastnosti *velocity*.

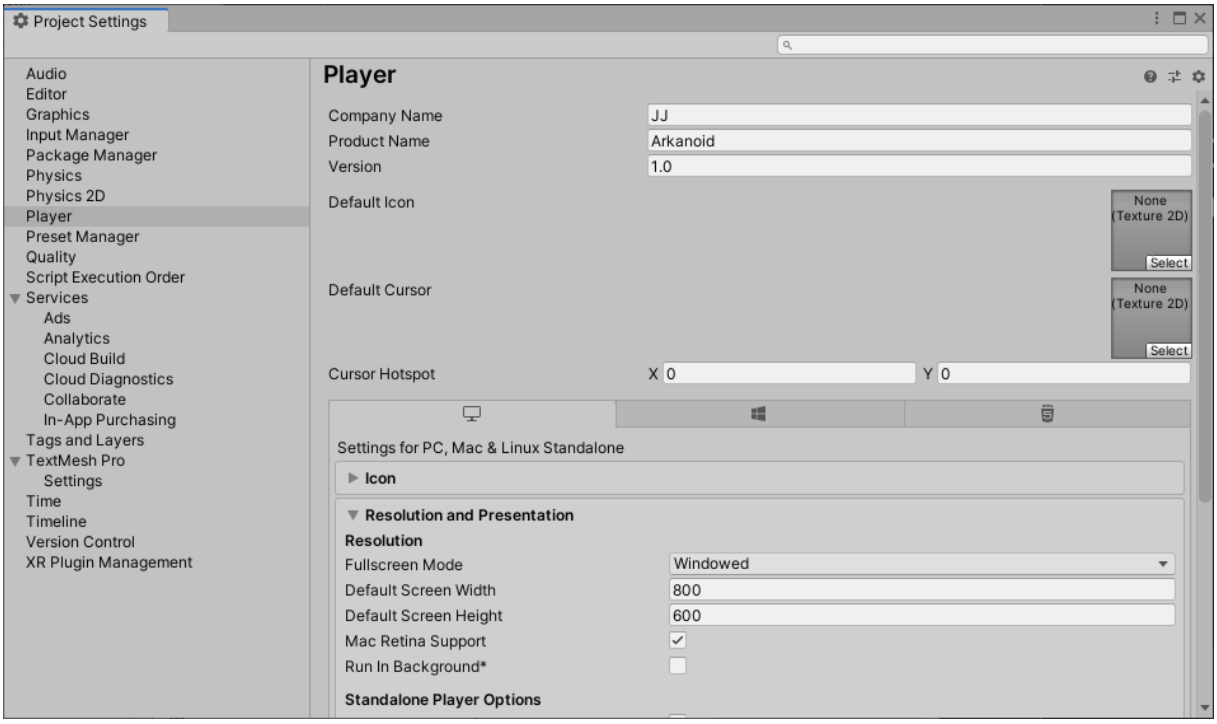
11.2 Ladenie hry (*tune*), testovanie hry (*playtesting*) a export (*build*)

Pred finálnym exportovaním (*build*) hry vždy odporúčame:

1. Odladiť hru tak, aby prinášala skutočný herný zážitok pre hráča.
2. Overiť aby každý level hry bol hrateľný napriek jeho možnej náročnosti, a tiež aby nebol príliš nudný.

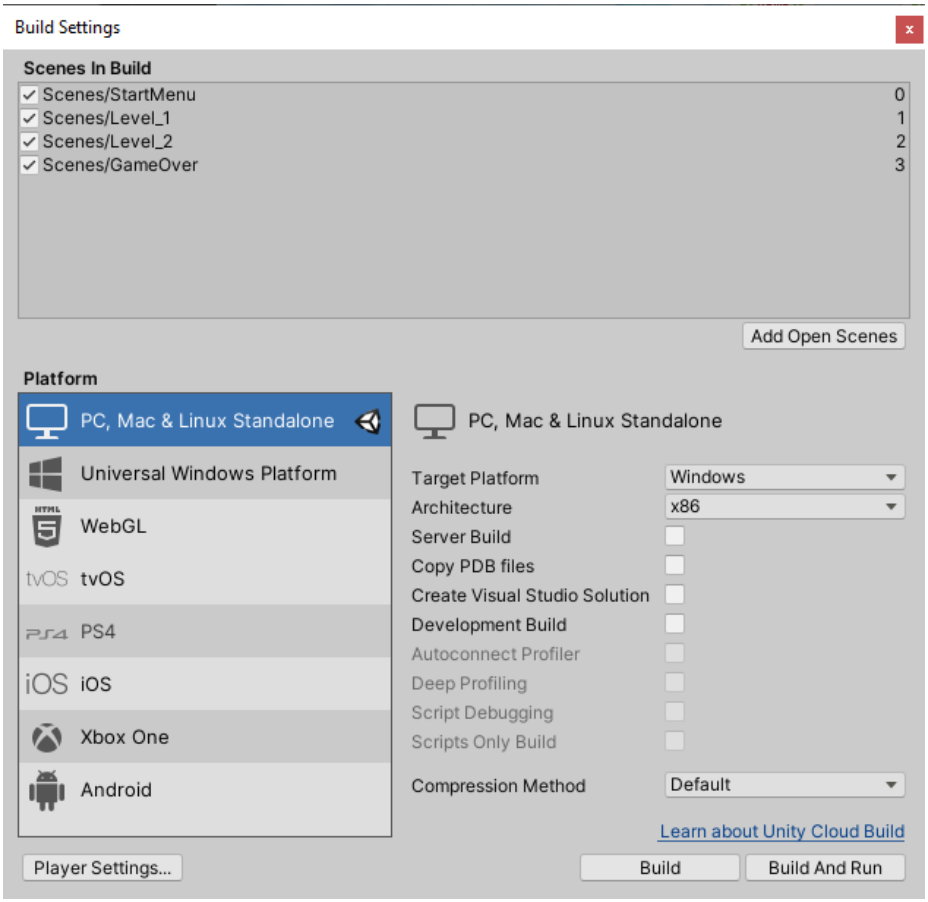
3. Skontrolovať všetky chyby (*bugs*), ako aj problémy.
4. Požiadať priateľov o zahratie hry a získanie spätnej väzby.

Pre výsledný export hry využijeme možnosť *File* → *Build Settings*, kde je možné okrem nastavenia cieľovej platformy nastaviť aj rôzne iné nastavenia. My exportujeme hru pre Windows a pomocou možnosti *Player Settings* v časti *Player* zvolíme *FullscreenMode* ako *Windoved*, pričom veľkosť výsledného okna je nutné nastaviť v pomere v ktorom sme hru vyvíjali. *Aspect Ratio*, ktorý sme pri vývoji hry používali bol 4:3 a rozlíšenie (*resolution*) 1440 x 1080. V prípade, ak chceme aby výška okna (*Default Screen Height*) bola 600, tak jeho šírka (*Default Screen Width*) musí byť 800 (hodnotu sme získali na základe výpočtu (600/4) * 3 = 800) (Obr. 210). Samozrejme, tieto, ako aj iné nastavenia exportu, sú plne na študujúcom.



Obr. 210 Nastavenia v časti *Player* pred výsledným exportovaním.

Potom už len stačí využiť možnosť *Build* (Obr. 211).



Obr. 211 Finálne exportovanie hry.

11.3 Kontrolné otázky a úlohy

1. Objasnite logiku zabezpečenia funkcionality automatizovaného testovania.
2. Ako je možné predísť nežiaducemu zacykleniu loptičky?
3. Aké nastavenia je možné ovplyvňovať pri exportovaní hry?
4. Rozšírte hru o personalizáciu, t. j. napríklad na začiatku hry vypýtajte meno hráča.
5. Rozšírte hru o systém bonusov na princípe náhodnosti, t. j. tieto budú vznikať pri rozbití bloku a budú sa aktivovať ak ich hráč zachytí padlom. Tieto môžu mať pozitívne, resp. negatívne účinky. Napríklad: zvýšenie počtu loptičiek; zmena veľkosti padla; zmena veľkosti loptičky; zmena rýchlosti; ovplyvnenie skóre; ovplyvnenie počtu životov; ovplyvnenie časového hľadiska a pod.

6. Rozšírte hru o možnosť ohodnotiť aktívnych hráčov, t. j. ak prídu napríklad hrať každý deň, tak mu na základe toho pridajte nejaký bonus a pod.
7. Rozšírte hru o časové hľadisko, t. j. môžete zaznamenávať čas hratia, alebo podmieniť prechod levelom, kým neuplynie nastavená časomiera. Respektíve, máte možnosť aj ovplyvňovať skóre hráča, ak stihne hráč dohrať level skôr, podľa toho koľko času ušetril, nastaviť výšku pridaného skóre, alebo zaviesť vizuálne hodnotenie levelov, ako býva zvykom, napr. hviezdičkami a pod.
8. Rozšírte hru o systém ukladania dát a aplikujte ho na rebríček najlepších hráčov so zaznamenaním údajov ako je meno, skóre/čas. Tiež využite systém na možnosť prerušenia hry, kedy pri opätovnom spustení bude mať hráč možnosť začať hrať odtiaľ kde skončil.

12 Zoznam použitej literatúry a informačných zdrojov

DAVIDSON, Rick and Ben TRISTEM. 2019. Complete C# Unity Game Developer 2D. [online]. [cit. 13.4.2020]. Dostupné na: ⟨https://www.udemy.com/course/⟩.

GREENO, James, 1994. Gibson' s Affordances. [online]. [cit. 23.6.2021]. Dostupné na: ⟨https://www.researchgate.net/publication/15176211_Gibson%27s_Affordances⟩.

iStock, 2021. Coronavirus Background Pictures, Images and Stock Photos. [online]. [cit. 3.1.2021]. Dostupné na: ⟨https://www.istockphoto.com/photos/coronavirus-background⟩.

NELSON, Mark, 2021. Breaking Down Breakout: System And Level Design For Breakout-style Games. UBM Tech, © 2021. [online]. [cit. 26.7.2021]. Dostupné na: ⟨https://www.gamasutra.com/view/feature/1630/breaking_down_breakout_system_and_.php?print=1⟩.

pixabay, 2021. Corona-emoji. [online]. [cit. 3.1.2021]. Dostupné na: ⟨https://pixabay.com/cs/⟩.

SINICKY, Adam. 2017. Learn Unity for Android Game Development: A Guide to Game Design, Development, and Marketing. United Kingdom: Apress. ISBN: 978-1-4842-2703-9.

STEWART, Samuel, 2020. Best Aspect Ratio For Gaming – Which Should I Choose? [online]. [cit. 8.5.2021]. Dostupné na: ⟨https://www.gamingscan.com/best-aspect-ratio-for-gaming/⟩.

Unity, 2020a. Working in Unity. [online]. [cit. 12.1.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/UnityOverview.html⟩.

Unity, 2020b. Canvas. [online]. [cit. 12.1.2021]. Dostupné na: ⟨https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html⟩.

Unity, 2020c. Text. [online]. [cit. 12.1.2021]. Dostupné na: ⟨https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/script-Text.html?q=Text⟩.

Unity, 2020d. TextMeshPro. [online]. [cit. 12.1.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/com.unity.textmeshpro.html⟩.

Unity, 2020e. Button. [online]. [cit. 19.1.2021]. Dostupné na: ⟨https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/script-Button.html⟩.

Unity, 2020f. Prefabs. [online]. [cit. 8.2.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/Prefabs.html⟩.

Unity, 2020g. Build Settings. [online]. [cit. 12.2.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/BuildSettings.html⟩.

Unity, 2020h. Camera. [online]. [cit. 18.2.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/class-Camera.html⟩.

Unity, 2020i. Colliders. [online]. [cit. 3.3.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/CollidersOverview.html⟩.

Unity, 2020j. Rigidbody 2D. [online]. [cit. 3.3.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/class-Rigidbody2D.html⟩.

Unity, 2020k. Physics Material 2D. [online]. [cit. 10.3.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/class-PhysicsMaterial2D.html⟩.

Unity, 2020l. Script Execution Order settings. [online]. [cit. 18.3.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/class-MonoManager.html⟩.

Unity, 2020m. Audio Overview. [online]. [cit. 18.3.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/AudioOverview.html⟩.

Unity, 2020n. Time.timeScale. [online]. [cit. 18.3.2021]. Dostupné na: ⟨https://docs.unity3d.com/ScriptReference/Time-timeScale.html⟩.

Unity, 2020o. Order of execution for event functions. [online]. [cit. 18.3.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/ExecutionOrder.html⟩.

Unity, 2020p. Particle systems. [online]. [cit. 7.4.2021]. Dostupné na: ⟨https://docs.unity3d.com/Manual/ParticleSystems.html⟩.

ZELENKA, Jakub, 2012. Shatter: Arkanoid nové generace. [online]. [cit. 26.7.2021]. Dostupné na: ⟨https://www.root.cz/clanky/shatter-arkanoid-nove-generace/⟩.

Názov:	VÝVOJ 2D PC HRY – ARKANOID V UNITY Návody na cvičenia (Skriptá a učebné texty)
Autor:	Ing. Jana Jurinová, PhD.
Recenzenti:	Ing. Miroslav Beňo, PhD. PaedDr. Patrik Voštinár, PhD.
Vydavateľ:	Univerzita sv. Cyrila a Metoda v Trnave
Rok:	2022
Rozsah:	161 strán/ 11,26 AH
Náklad:	50 ks
Grafická úprava obálky:	Ing. Jana Jurinová, PhD.
Tlač:	elektronická verzia