Tomáš Holan

Unity

První seznámení s tvorbou počítačových her

Edice CZ.NIC

UNITY
První seznámení s tvorbou počítačových her

Tomáš Holan

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2020
Kniha vyšla jako 26. publikace v Edici CZ.NIC.
ISBN 978-80-88168-60-7

CZ.NIC | SPRÁVCE DOMÉNY CZ

ISBN 978-80-88168-60-7

— Tomáš Holan

Unity

První seznámení s tvorbou počítačových her

— Edice CZ.NIC

Předmluva vydavatele

Předmluva vydavatele

Vážení čtenáři,

dostává se vám do ruky tak trochu jiná kniha o programování. Odborné publikace se většinou pyšní hloubkou, do které zasahují, a objemem informací, které obsahují. Tato kniha naopak sází na čtivost, praktičnost a obecně zábavnost. A hned v úvodu otevřeně přiznává, že spousty témat se ani nedotkne.

S programováním jsem začínal na základní škole před skoro třiceti lety. A jedna z mých prvních myšlenek samozřejmě byla napsat si hru. Tehdy jsem zvládnul nanejvýš jednoduchý kvíz, ale to mi nebránilo snít. Postupem času jsem se zlepšoval a učil se toho víc a víc, ale začal jsem řešit reálné problémy, a na hry už nebyl čas. Když jsem s určitou technologií začínal, čerpal jsem obvykle z nějaké chytré knihy, která hlásala, že obsahuje vše, co potřebuji vědět. Já se ponořil do jejího čtení a nikdy ji nedočetl, protože po pár kapitolách suché teorie mě to přestalo bavit.

Tato kniha je jiná. Netvrdí, že vás naučí vše o Unity. Právě naopak, hrdě se hlásí k tomu, že neobsahuje ani solidní základy, natož kompletní popis celého enginu. Její předností je něco jiného, podle mě daleko zásadnějšího. Je velmi čtivá a názorná. Ukazuje, jak začít. Začne jednoduchým zadáním a snaží se ho naplnit. Vysvětluje, co je potřeba k dosažení cíle. Nestrávíte kapitoly čtením teorie, abyste se na konci knihy dozvěděli, že vám to možná někdy k něčemu bude. Naopak. Kniha začíná od toho, co je naším cílem, a pak teprve řeší, jak ho dosáhnout. Případně jak cíl změnit. Protože v reálném světě neexistuje perfektní zadání. Existují celé vývojářské týmy, které pracují roky na funkcích, které se nakonec ukáží jako okrajové a zbytečně složité, a jsou zahozeny. A přitom by mnohdy stačilo se nad zadáním včas zamyslet a upravit ho. V knize naleznete momenty, kdy se fungování hry změní, protože naplnění původního plánu by bylo zbytečně složité a nakonec by ani nemuselo fungovat dobře. Stejně jako v reálném životě do sebe v knize vše nezapadá na první pokus. Objevují se problémy, které je třeba řešit, a kniha ukazuje jak. I díky tomu vás „vtáhne do děje“.

Pro koho je tedy kniha určena? Není to publikace, kterou bude mít programátor her na poličce, a po které sáhne, když nebude vědět, jak se nějaká obskurní konstrukce používá. Je to kniha, kterou zhltnete, pokud si chcete naprogramovat nějakou hru a nevíte, jak začít. A nemusí být jen pro ty, kteří už umí programovat. Ukázky jsou velmi názorné a jednoduché. Osobně mě překvapilo, jak málo a jak jednoduchý kód vlastně stačí k tomu, udělat si vlastní hru. Může to tak být i vhodná kniha pro všechny, kteří přemýšlí, jak začít s programováním vůbec. Za málo práce hodně zábavy. A jak s tím jednou začnete, už vás to jen tak nepustí.

Přeji vám zábavné čtení a těším se na spousty nových Indie her, u jejichž zrodu bude právě tato kniha stát.

Michal Hrušecký, CZ.NIC

Obsah

1 Úvod

1 Úvod

1.1 Procházka

Tohle je kniha o programování her v Unity enginu a je určená těm, kdo by se chtěli s Unity seznámit, ale nevědí, odkud začít.

Nebude to učebnice, kde bych vyjmenovával všechna okna, tlačítka a položky v menu. Budu psát program a co budu potřebovat, to použiju a okomentuju. Taková společná procházka, možná trochu upovídaná.

Expert

Nejsem žádný expert na Unity, na druhou stranu programuju více než čtyřicet let a leccos z toho, co jsem programoval, byly hry, tak věřím, že mám trochu nadhled a že to půjde. A že pokud mi nějaká znalost bude chybět, tak si ji dokážu najít. Vzhůru do toho!

Varování

Budu ukazovat vývoj programu nebo programů a to zahrnuje i změny plánů a nesplněná předsevzetí. Mohl bych se jim vyhnout a ukázat vám jen výsledek, kde všechno pěkně vychází, ale připadá mi, že to bloudění a přehodnocování k vývoji programu patří. Navíc mi to dovolí ukázat víc možností, jak něco udělat, než jen jedinou správnou. A nakonec, chceme se něco naučit, takže cesta je cíl!

Hry

Když říkám hry, tak tím nemyslím šachy ani Hledání min, i když to by šlo v Unity naprogramovat taky. Půjde mi o hry, ve kterých se něco hýbe, chodí, létá, padá, plave.

Herní smyčka

Takové hry mají v sobě vlastně model celého herního světa – údaje o tom, kde je jaká postavička překážka, věc… a celá hra potom probíhá v něčem, čemu se historicky říká herní smyčka, tedy neustálé opakování dvou kroků:

- pohni všemi objekty světa
- vykresli svět

Herní objekt

Ano, ty postavičky, překážky, věci… prostě všechno, co ve hře má nějaký význam, se shrnuje pod označení **herní objekt.**

1.2 Unity engine

Cizí kód

Fred Brooks v roce 1987 napsal článek *„No Silver Bullet – Essence and Accident in Software Engineering"*, kde se zamýšlí nad tím, jestli něco dokáže v průběhu deseti let řádově urychlit vývoj software (nebo ho urychlovat tak, jak se zrychluje výkon hardware) – a dochází k závěru, že ne.

Softwarové projekty už tehdy měly zpoždění, nedodržovaly rozpočet a trpěly podobnými neduhy. Článek postupně prochází možnosti, které by mohly pomoci, včetně automatického programování (pamatuju, byl to hit) nebo objektového programování (to už dnes používá snad každý) a postupně dochází k tom, že žádná z těch nových technik nedokáže řádově urychlit vývoj programů, prostě proto, že vývoj programů je v principu složitý (to je ta „essence" z názvu článku).

Ale při pohledu zpět vidíme, že vývoj programů dnes rychlejší je; třeba program napodobující kapesní kalkulačku, s tlačítky a ovládáním myší dokáže student prvního ročníku vysoké školy napsat za hodinu, možná méně. A je to díky možnosti, kterou autor článku nezapočítal – možnosti použít cizí kód! (Slovem kód tady označuji i dále budu označovat program, celý nebo část, ve zdrojovém tvaru nebo nějak přeložený.)

Když zmíněný student píše program ovládaný myší, tak nemusí řešit komunikaci s myší, ať už pomocí sériového rozhraní RS232 (dříve) nebo pomocí USB (dnes), protože o to se stará operační systém. A pokud jiný student chce ve své webové stránce vykreslovat trojrozměrné animace, použije na to nějakou knihovnu (nebudu jmenovat, ale je jich víc).

Používání cizího kódu s sebou nese spoustu problémů, ale o tom teď nechci mluvit.

Knihovna, framework, engine

Knihovna je jedna z možností, jak sdílet a znovu-používat dříve napsaný kód.

Většina programovacích jazyků umožňuje do zdrojového kódu začlenit další kód, buďto ve formě zdrojového kódu nebo výsledek překladu a tak stačí příslušné funkce (případně data, definice tříd a podobně) vložit do nějakého zdrojového kódu nebo projektu nebo balíčku (různé názvy v různých programovacích jazykách), který budeme dále používat; možná my a možná i někdo

jiný. Takovému (zdrojovému) kódu nebo množině (zdrojových) kódů potom říkáme knihovna. Z knihovny si vytáhneme, co se nám hodí, a také můžeme klidně použít více knihoven najednou, třeba jednu pro kreslení a druhou pro komunikaci po internetu.

Framework funguje v jistém smyslu naruby – zatímco u knihovny „hlavní program" píšeme my a vybíráme si, co použijeme z cizích knihoven, u frameworku je to tak, že funguje sám a my pouze můžeme doplňovat nebo měnit některé funkce. Třeba když v .NET Frameworku vytvoříme WinForms-aplikaci, nemusíme napsat ani řádku kódu a přesto máme program, který jde spustit, zobrazí okno, které se dá zvětšovat a přesouvat myší i klávesami, zobrazuje svou ikonku v seznamu běžících programů, zobrazuje se ve správci úloh a má řadu dalších funkcí. A pokud chceme, můžeme pozměnit jeho funkci, pomocí které na začátku vytváří obsah svého okna a přidat tam nějaká tlačítka nebo jiné komponenty a můžeme také přidat funkce, které budou reagovat na události, a tím vlastně zařídit, že ten program bude dělat to, co chceme. Ale hlavní program jsme měli od samého začátku hotový a jenom doplňujeme a vylepšujeme. A hlavní program jsme nenapsali my, ale autoři frameworku.

Herní engine (a Unity není jediný (pamatuje někdo AGI? Třeba King's Quest od Sierra Online?), viz třeba Unreal Engine, Cry Engine nebo Godot) je povahou spíš framework.

Obvykle zahrnuje **vykreslování** (protože to je spousta práce a herní návrháři mají jiné zájmy), **detekci kolizí** (abychom poznali, kdy na sebe objekty narazily), **fyziku** (aby na objekty mohly působit síly, setrvačnost a další objekty), **herní smyčku** plus nástroje na vytváření a editaci herního světa přidáváním a nastavováním herních objektů. Většina herních enginů dovoluje také programování, ať už ve svém vlastním jazyku nebo v některém z univerzálních jazyků. V Unity lze programovat v jazyku C#.

Dostupnost, licence

Unity engine je ke stažení ze stránek společnosti, dostupná je jak bezplatná, tak placená verze. Bezplatná verze je omezená pro osobní použití a pro společnosti se ziskem do určité výše.

Verze

Jako každý software, který se používá, i Unity engine se vyvíjí. To znamená, že pokud si stáhnete nejnovější verzi, za čas už nebude nejnovější. Novější verze může být lepší (proto to dělají), ale Unity nemusí umět bez problémů otevřít projekt, který jste vytvořili v jiné verzi (viz ta moje poznámka o problémech s používáním cizího kódu). Takže (obzvlášť pokud chcete vytvářet nějakou hru společně s dalšími lidmi) je důležité domluvit se, jakou verzi Unity budete používat.

1.3 Zdroje informací

Návodů, jak v Unity udělat to či ono, je plný internet. Nejspolehlivější, nejpečlivější, nejudržovanější a další nej- je **Unity Manual** od autorů Unity na stránce https://docs.unity3d.com/Manual/, nezapomeňte si vlevo nahoře vybrat verzi Unity, o které chcete informace. Můžete si ho i stáhnout jako zip.

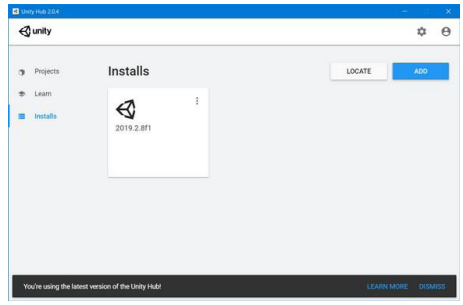
2 Začínáme

2 Začínáme

2.1 Začínáme

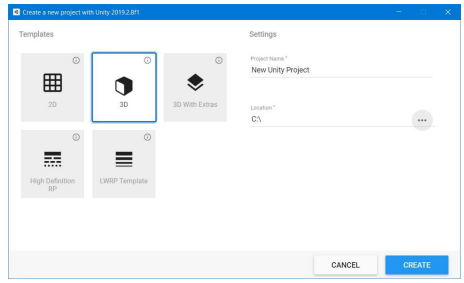
V téhle kapitole se chci podívat na to, jak vytvořit nový projekt a jak vypadá pracovní prostředí. Minimální cíl je vytvořit něco, co se bude alespoň trochu hýbat.

Když začínám, tak bych si měl rozmyslet, jakou verzi Unity chci používat. Unity už několik let disponuje nástrojem **Unity Hub**, pomocí kterého můžete mít přehled o všech projektech, ale také o všech instalovaných verzích Unity (ano, lze mít nainstalováno několik verzí současně). Unity Hub můžeme použít k instalaci nových verzí a vytváření nových projektů. A v neposlední řadě pod záložkou **LEARN** nabízí ke stažení ukázkové projekty i odkazy na další informace.



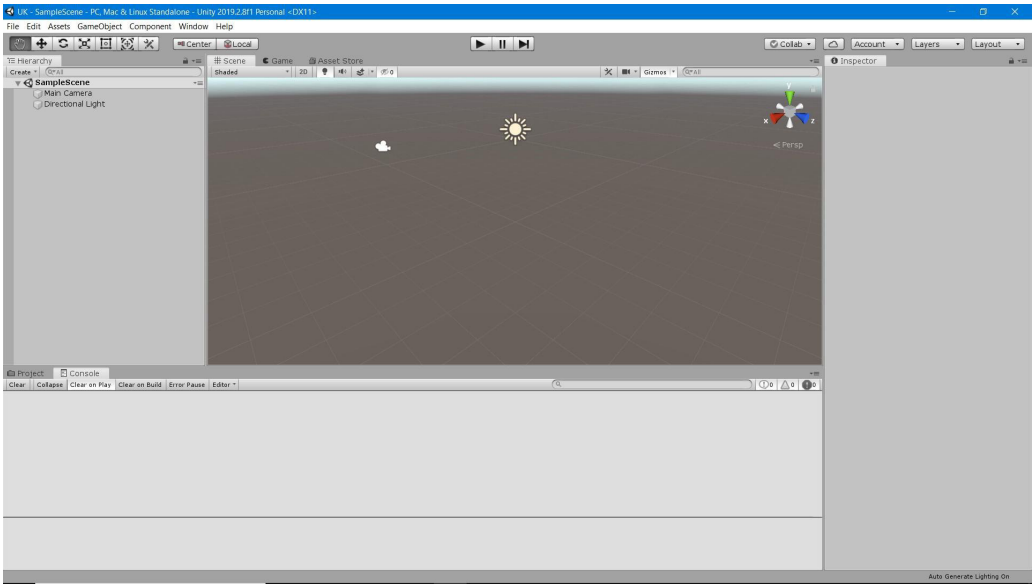
V tuto chvíli mám nainstalovanou verzi **2019.2**, ale protože verze 2020 jsou dostupné teprve jako pre-release, zůstanu u ní.

Jdu tedy vytvořit nový projekt a budu používat **Unity 2019.2.** Na záložce **Projects** mačkám tlačítko **NEW** a když si chci vybrat kliknutím na sousední šipku dolů, nabízí výběr, kterou verzi Unity chci použít, mám jedinou, tak žádné rozhodování. Na to vyskočí okénko, kde si vybíráme typ projektu, název a umístění na disku:



Nastavuju jméno **UK**, neuváženě mačkám **Enter** a tím startuje Unity. Budu mít projekt umístěný v kořeni disku **C:**. Start Unity chvíli trvá, nenechte se tím znepokojit.

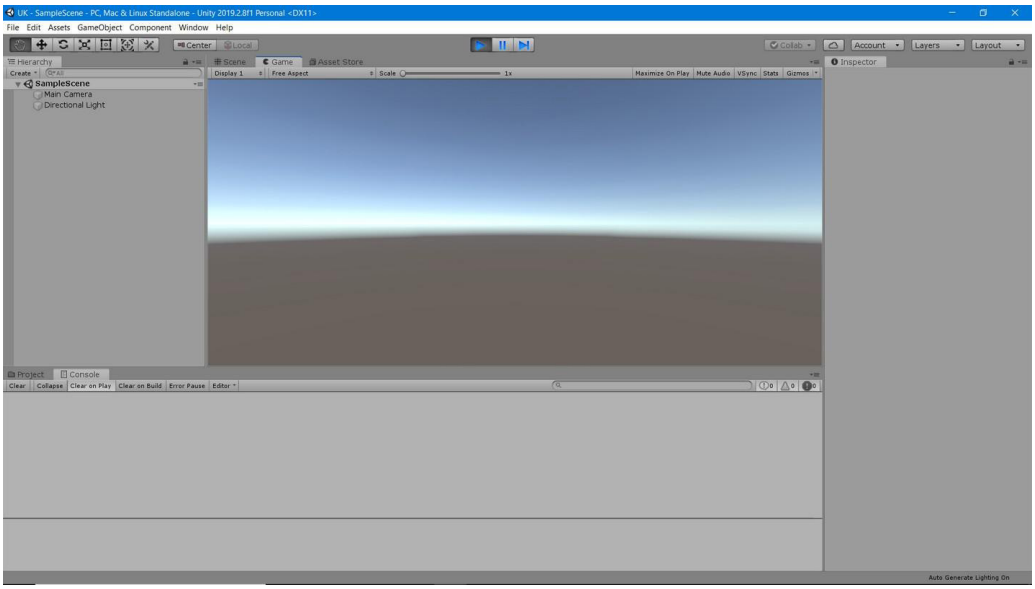
2.2 Prostředí Unity



Víme, že Unity slouží k vytváření her, víme, že hra obsahuje nějaké herní objekty, které se opakovaně (vnucuje se „pravidelně“, ale to není pravda!) pohybují a vykreslují. To, jak se budou objekty vykreslovat, určuje **kamera**, aby bylo (kamerou) něco vidět, potřebujeme **světla**, a tomu celému se říká **scéna**. Takže okno **Hierarchie (**vlevo) zobrazuje **strukturu scény** (scén může být v projektu více, hra má obvykle kromě vlastního hraní i nějaký začátek, konec a další části), to velké okno uprostřed zobrazuje **editor scény**, zatím obsahuje pouze kameru a jedno světlo, okno **Inspektor** napravo zobrazuje vlastnosti vybraného objektu a panel dole zobrazuje **konzoli** nebo další **Assets**, později.

A nahoře vidíme tlačítka **Play**, **Pause** a **Step** pro spuštění hry – tak zkusíme stisknout **Play**.

Na to uvidíme pohled kamerou:



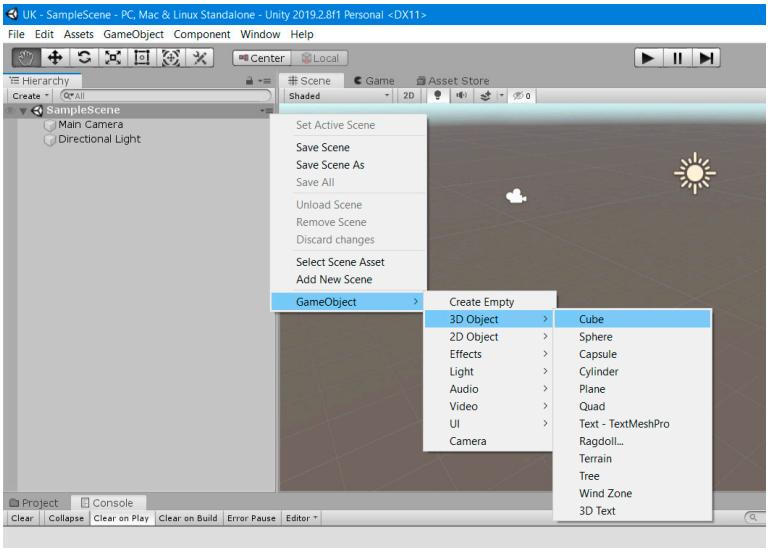
a jinak nic zajímavého, protože v naší hře zatím nejsou žádné herní objekty. Běh ukončíme opětovným stisknutím tlačítka **Play** (nebo klávesovou zkratkou **Ctrl-P**).

Mohli jsme si také všimnout, že se změnila vybraná záložka nad obrázkem hry – zatímco předtím byla vybraná záložka **Scene**, během běhu byla vybraná záložka **Game**. Když se budeme chtít někdy podívat, jak vypadá naše scéna z pohledu kamery, nemusíme hru spouštět a stačí se přepnout záložkou.

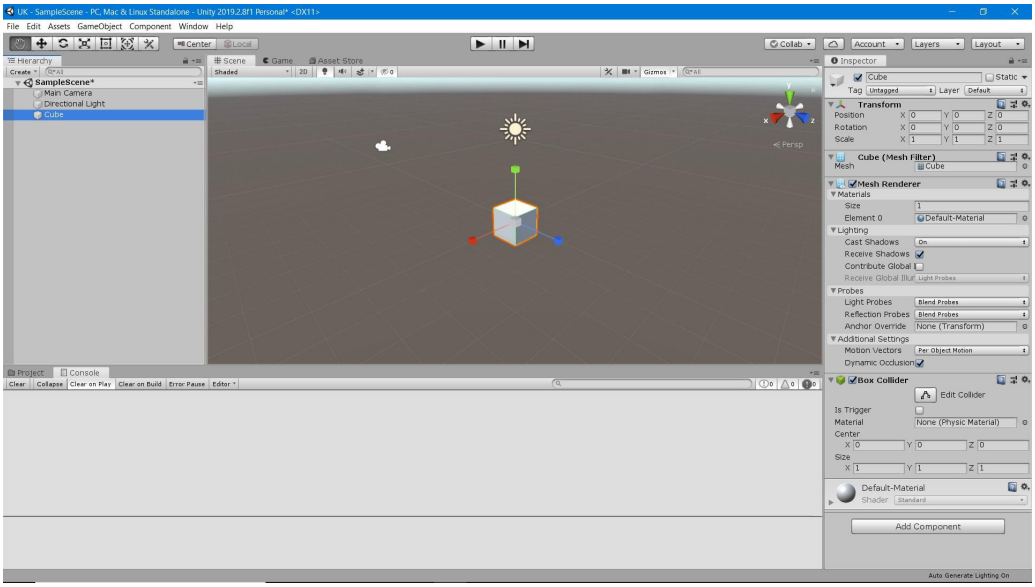
Přidat herní objekty

Pojďme ale do naší hry přidat nějaké objekty. Kdybychom programovali opravdovou hru, asi bychom si objekty nejdříve vytvořili-vymodelovali, třeba v programu Blender, spousta objektů se také dá koupit nebo bezplatně stáhnout, ale protože nechceme odbočovat, použijeme jednoduché geometrické objekty, které jsou v nabídce Unity.

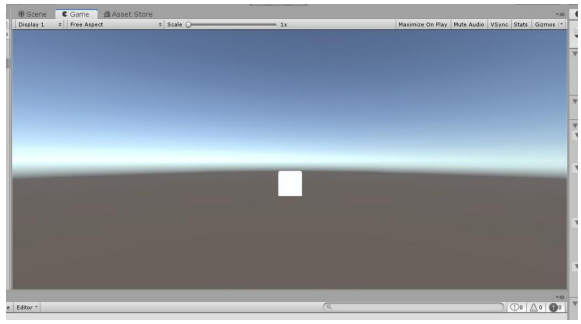
Když klikneme na ikonku v pravé části okně Hierarchie, vyskočí nám menu, kde si vybereme, že chceme do scény přidat krychli:



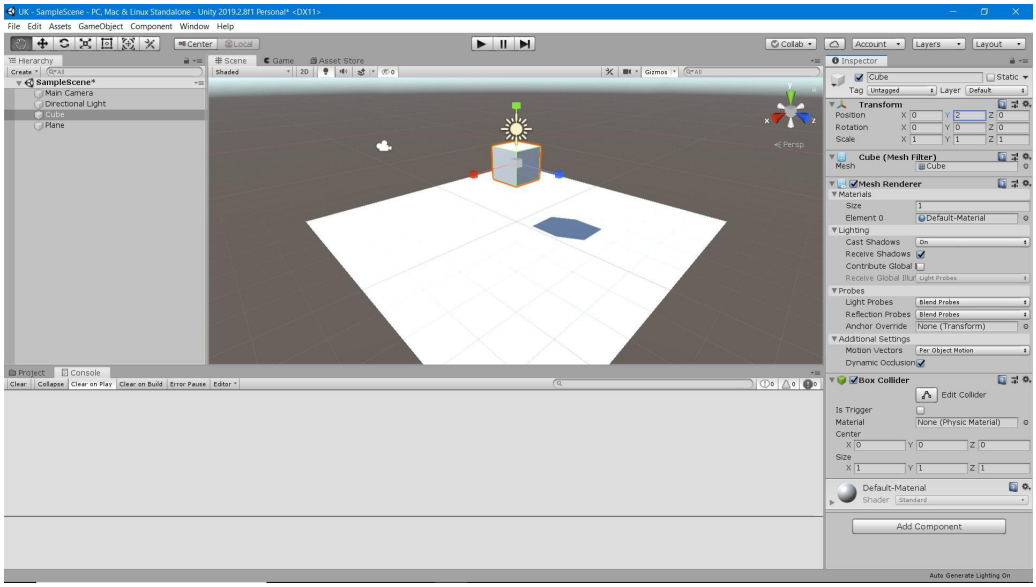
a naše scéna od téhle chvíle obsahuje krychli, v okně Hierarchie i v editoru scény. A protože je vybraná, můžeme vidět a měnit její vlastnosti v okně Inspektoru napravo.



Pokud se podíváme na pohled kamerou, vidíme naši krychli jako bílý čtverec uprostřed záběru:

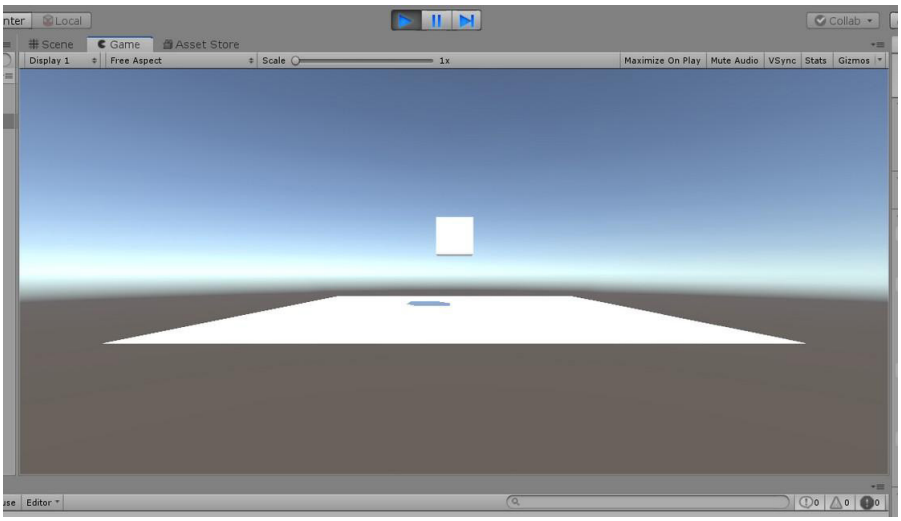


Přidejme si do scény ještě jeden objekt, tentokrát to bude rovina (**Plane**), kliknutím si vybereme krychli a v inspektoru v části **Transform** jí změníme Y-složku údaje **Position** z 0 na 2:



Můžeme si všimnout, že kostka vrhá stín.

Pokud teď spustíme hru tlačítkem **Play…**



...kromě přepnutí na pohled kamerou se neděje nic. No, nic jsme nenaprogramovali, co by se mělo dít… ale možná někdo přece jen v koutku duše doufal, že uvidí něco pěkného…

2.3 Fyzika

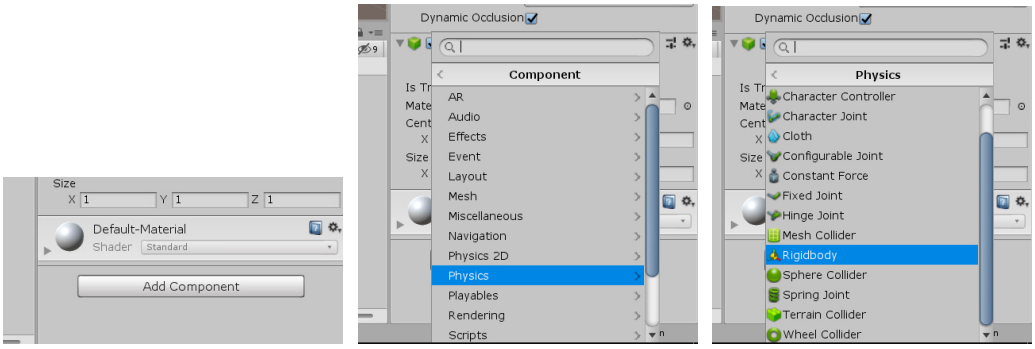
Jedna z úloh, které herní enginy řeší za autory her, bývá detekce kolizí a „fyzika", tedy to, že objekt může mít nějakou rychlost, nějakou hybnost, nějakou pružnost a pokud by měl při pohybu projít jiným objektem, tak neprojde, případně se odrazí nebo jinak změní svou rychlost, případně se otočí a podobně.

Tohle všechno Unity umí, jen si musíme říct, že to chceme.

Věšák na vlastnosti

Herní objekty v Unity samy od sebe neumí nic a slouží jako věšáky, na které navěšujeme další vlastnosti, kterým se v Unity říká **komponenty**. Když jsme pomocí menu do naší scény přidali krychli a rovinu, tak ty už měly některé komponenty nastavené, takže v Inspektoru můžeme vidět, že naše krychle obsahuje komponentu **Mesh Renderer**, která se stará o vykreslování a komponentu **Box Collider**, která řeší kolize hranolu, na rozdíl od komponenty **Mesh Collider**, kterou obsahuje naše rovina.

Pokud tedy chceme (chceme!), aby se krychle pohybovala (padala), potřebujeme přidat komponentu, která se o to bude starat: **Add Component** - **Physics** – **RigidBody**:



Když teď stiskneme **Play**, kostka spadne a zarazí se o objekt roviny, protože oba mají komponentu pro detekci kolizí.

Vylepšení

Celý ten pád trvá jen chviličku a na tom, jak výkonný máte počítač, zaleží, kolikrát se během té doby scéna vykreslí. Pokud si ho chceme trochu víc užít, můžeme do scény přidat ještě jednu krychli, trochu výš a trochu stranou, aby nezůstala stát na první krychli, ale padala dál:

Její **Position.Y** nastavíme na **5**, přitom nám krychle zmizí z okna editoru, ale stačí zazoomovat kolečkem myši – a **Position.X** nastavíme na **0.9** (jako desetinný oddělovač můžeme použít čárku i tečku), aby nová krychle při pádu jen zavadila o starou krychli.

Pokud bychom teď stiskli tlačítko **Play**, nová krychle zůstane na místě, protože jsme jí nenastavili, že má padat, takže ještě přidat komponentu **Rigid Body** – a padá a kutálí se!

Takže máme první projekt a hýbe se to!

2.4 Vlastnosti, parametry atributy

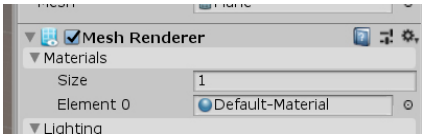
Vzhled i chování objektů a komponent můžeme ovlivnit nastavením jejich parametrů. V samotném prostředí Unity to můžeme udělat pomocí panelu Inspektor, ale až začneme programovat, můžeme vlastnosti nastavovat i dosazením ze skriptu (kódu, který bude součástí hry).

Materiál

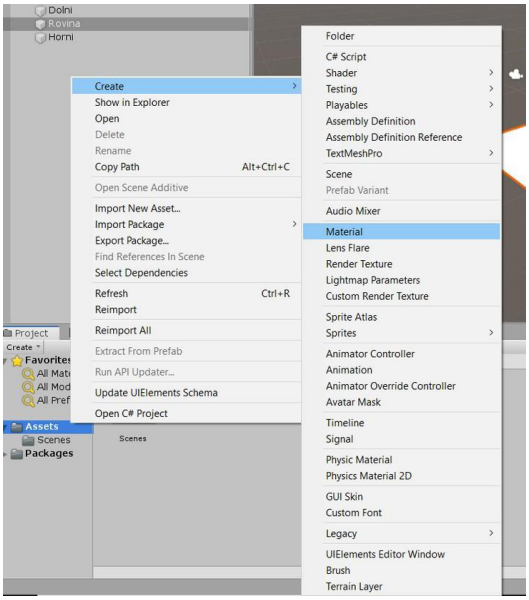
Začněme tím, jak naše dvě krychle vypadají. A ten začátek ještě začněme tím, že je přejmenujeme, protože teď se jmenují **Cube** a **Cube (1)**. Přejmenování je snadné, stačí v panelu scény vybrat ob-

jekt, stisknout F2 a změnit jméno. Pojmenoval jsem je **Dolni** a **Horni**, bez diakritiky. Analogicky jsem rovinu **Plane** přejmenoval na **Rovina**.

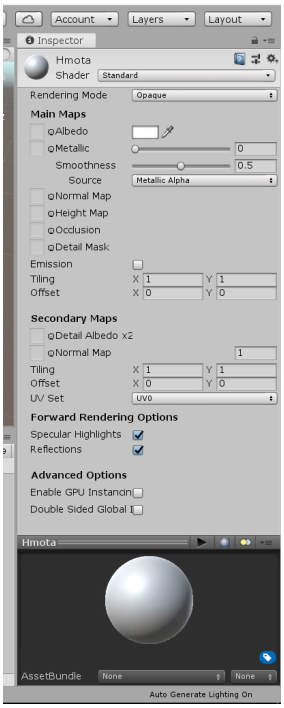
Všechny tři objekty jsou bílé, když se podíváme do Inspektoru, vidíme, že Rovina používá výchozí materiál.



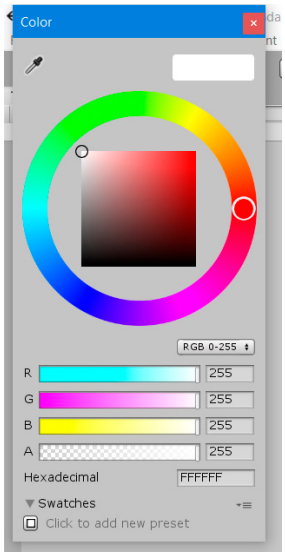
Když klikneme na ikonku napravo od názvu materiálu, vyskočí nám okénko pro výběr materiálu, ale my si nejdříve vytvoříme vlastní materiál. V dolním panelu, přepneme na záložku **Project** a na položce **Assets** (to jsou všechny „věci" patřící k projektu – obrázky, zvuky, skripty – a také materiály) pravým tlačítkem vyvoláme menu a zvolíme **Create – Material**:



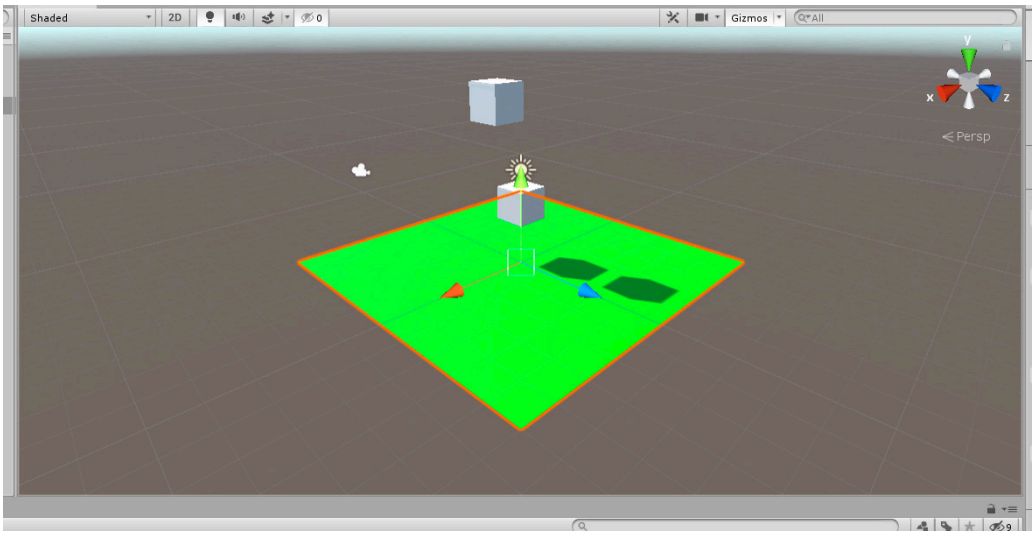
Nový materiál hned přejmenujeme na **Hmota** (nepíšu mezinárodní program a české jméno na první pohled odliší moje prvky od standardních) a protože je vybraný, tak Inspektor zobrazí jeho vlastnosti:



, kde vidíme, že můžeme nastavit vlastnosti jako průhlednost nebo lesk – nebo barvu! Bílý obdélníček napravo od vlastnosti **Albedo** při kliknutí vyvolá okno pro výběr barvy:



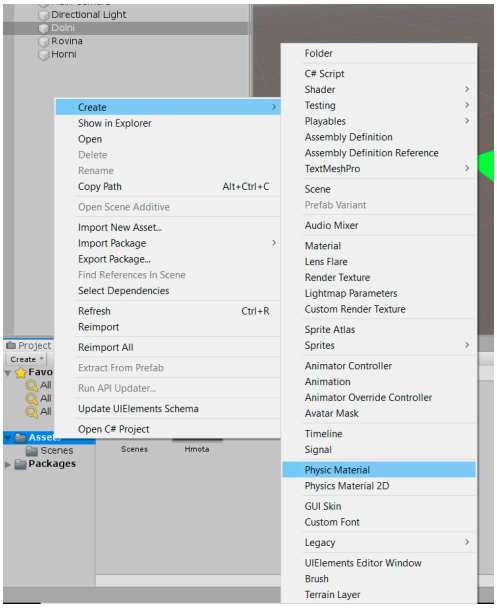
Abychom nastavené vlastnosti přenesli na objekt, vybereme ho, a buďto v Inspektoru u položky Material kliknutím vyvoláme nabídku materiálů a vybereme si nebo (snazší) požadovaný materiál v dolním okně chytíme a přetáhneme na vlastnost **Material** nebo přímo na objekt ve scéně:

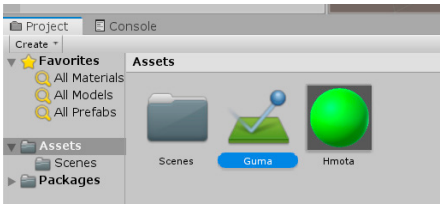


Bylo by samozřejmě lepší nový materiál nevytvářet takhle v kořeni přihrádky **Assets**, ale udělat si zvláštní přihrádku na materiály, ale pro začátek nám to nevadí.

Fyzikální materiál

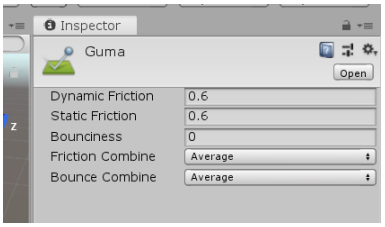
Materiál, který jsme si vytvořili v minulé kapitole, nám pomohl změnit vzhled. To odpovídá tomu, že jsme tento materiál použili v komponentě **Mesh Renderer**, která má na starosti zobrazování. Umíme si ale představit i materiál, u kterého není důležitý vzhled, ale fyzikální vlastnosti, třeba pružnost. Takový materiál si zase můžeme vytvořit a přidat do Assets:





Nastavíme mu jméno, třeba **Guma** a zkusíme změnit jeho vlastnosti tak, aby pružil.

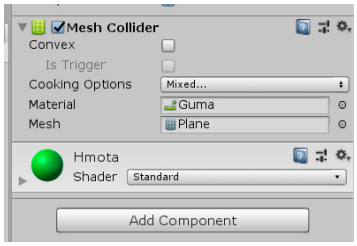
Když si materiál **Guma** vybereme, můžeme v Inspektoru nastavit jeho vlastnosti



a změníme hodnotu vlastnosti **Bounciness** na **1**.

Protože odrazivost (stejně jako tření) záleží na odrazivosti obou střetávajících se objektů, můžeme nastavit, jak se tyto vlastnosti mají kombinovat. Zatím ponecháme hodnotu **Average** čili průměr a přiřadíme tento materiál objektu **Rovina**, opět přetažením materiálu na objekt.

Výsledný pohled na komponentu **Mesh Collider** objektu **Rovina** vypadá takto:



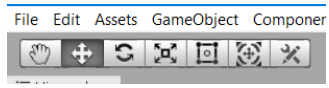
Když stiskneme tlačítko **Play**, vidíme, že kostičky dopadají jinak. Ale přidáme fyzikální materiál **Guma** ještě zbylým objektům **Horni** a **Dolni** a teď kostičky skáčou tak dlouho, dokud jedna z nich nepřepadne přes okraj **Roviny** a nezřítí se dolů. Odrazivost **1**, tedy 100 procent, je přece jen příliš.

2.5 Kamera

Pokud chceme změnit pohled na naši scénu, upravíme nastavení kamery.

Jakmile kameru vybereme, objeví se nám v okně scény pod-okénko Camera Preview, kde můžeme vidět, jak vypadá pohled vybranou kamerou. V Inspektoru můžeme nastavovat vlastnosti kamery jako způsob projekce (Projection) nebo co se má zobrazovat tam, kde není nic (Clear Flags).

My ale chceme kameru spíše posunout výš a pootočit, k tomu slouží tlačítka pod lištou menu použitelná pro manipulaci se všemi objekty,



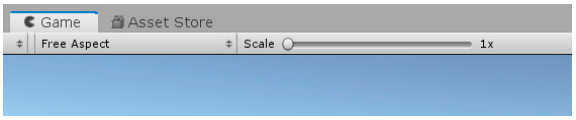
kterými postupně ovládáme posunutí pootočení, zvětšování a další.

Takže chceme-li kameru posunout výš a potom pootočit tak, aby směřovala do středu objektu **Rovina**, můžeme na to použít nástroj pro posouvání a nástroj pro pootáčení, případně sdružený nástroj druhou ikonkou zprava.

Po chvíli možná skončíme u toho, že vidíme podivně křivý pohled, který se nám nedaří srovnat, takže opustíme nástroje a pozici a úhly natočení kamery nastavíme v okně Inspektoru tak, že do vlastností (X, Y, Z) **Position** dosadíme hodnoty **0**, **4**, **-10** a do vlastností **Rotation** hodnoty **15**, **0**, **0**.

Poznámka: *Asi vás už napadlo, že když nastavujeme směr pootočením podle tří os, záleží na tom, v jakém pořadí se ta pootočení budou provádět (jestli vás to nenapadlo, tak si to zkuste představit). Unity takto zadané směry pootáčí* ***postupně podle osy X, podle osy Y a podle osy Z, v tomto pořadí****.*

Může se nám u toho stát, že pohled v okénku náhledu **Camera Preview** bude vypadat jinak než pohled po přepnutí na záložku **Game** nebo ve vlastní hře. Pokud se to stalo, přepněte se na záložku **Game** a posuvník **Scale** posuňte až na levý okraj:



2.6 Konec začátku

Pokud jsme došli až sem, měli bychom mít hrubou představu o tom, jak se s Unity pracuje, i když většina možností byla popsaná nepořádně a celá řada jich nebyla ani zmíněna. Ale na ně se podíváme v dalších částech.

3 Pokračujeme

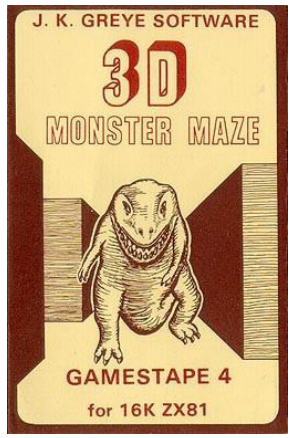
3 Pokračujeme

3.1 Návrh hry

V minulé části jsme vytvořili něco, co se nakonec zobrazovalo a chvíli pohybovalo a na čem jsme mohli vidět základní nástroje, kterými disponuje Unity. V této části bychom se chtěli věnovat dalším nástrojům i dalším možnostem těch nástrojů, které jsme už viděli a na to bychom potřebovali něco složitějšího, než jen dvě kostičky, které padají a skáčou. Tak si pojďme něco navrhnout.

Tyranosaurus Rex

V osmdesátých letech byl populární počítač **Sinclair ZX81**, alespoň do doby, než přišel jeho ještě populárnější následník **Sinclair ZX Spectrum.** ZX81 měl 1kB paměti a dal se k němu připojit modul s dalšími 16kB a na těchto 17 kB paměti fungovala hra **3D Monster Maze**, ve které hráč procházel trojrozměrným bludištěm a snažil se nepotkat výše uvedenou obludu.



(zdroj: https://en.wikipedia.org/wiki/File:3DMonsterMaze.JKGS.tape-cover.jpg)

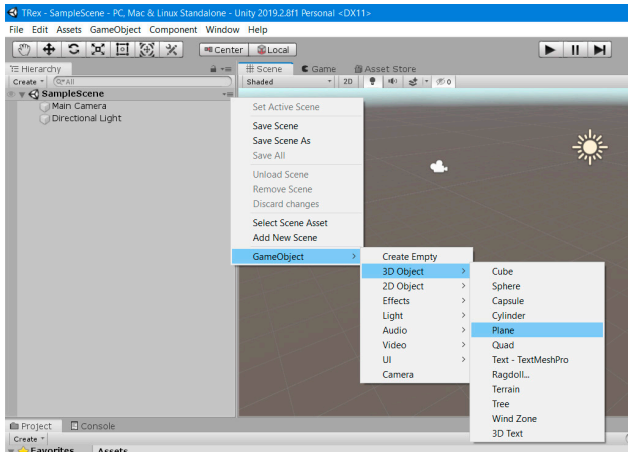
Kdybychom chtěli takovou hru naprogramovat dnes, navíc pomocí Unity, tak budeme potřebovat:

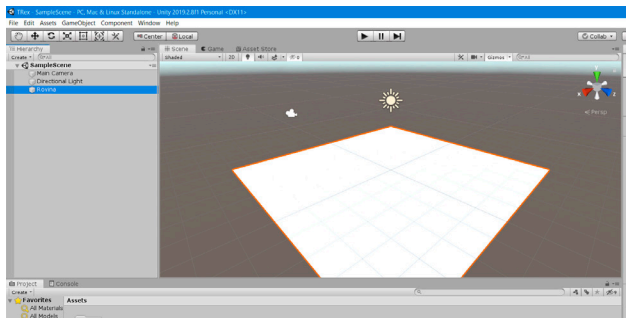
A) hrdinu a jeho pohyb pomocí šipek nebo myši
B) bludiště, které bude hráče omezovat v pohybu
C) kameru, která bude sledovat hrdinu
D) tyranosaura, který bude buďto setrvávat na jednom místě nebo se bude pohybovat a tedy bude mít nějakou umělou inteligenci (AI)
E) něco, co bude hráč hledat, buďto východ nebo nějaké bonusy, aby měl důvod se v bludišti pohybovat
F) zvuky?
G) Světlo, kterým bude svítit hrdina?
H) Vytváření bludiště programem, aby nebylo vždycky stejné?
I) ještě něco, co mě teď nenapadá.

3.2 Nový projekt

Vytvoříme si nový projekt. Použijeme na to **Unity Hub**, typ projektu bude **3D**, název **TRex** a cesta tam, kde ho chceme mít uložený. Projekt vytvoříme tlačítkem **CREATE**.

Do projektu zase vložíme objekt **Plane** a pojmenujeme ho **Rovina**:





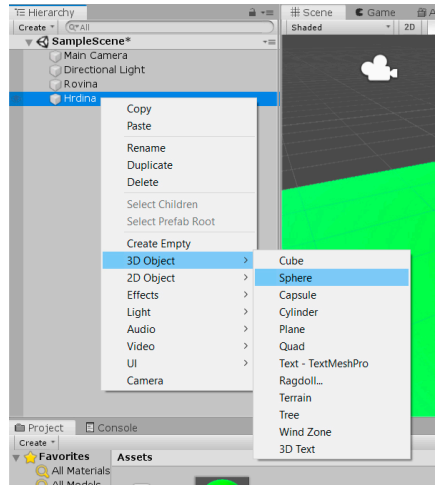
3.3 Pohyb, skripty, ovládání

Do hry potřebujeme něco, co bude representovat hráče. U toho bychom se měli rozhodnout, jestli hrdina bude **viditelný** nebo ne; **neviditelný** hrdina není žádné kouzlo, v některých hrách se hráč dívá z pohledu hrdiny a tedy hrdina sám není vidět.

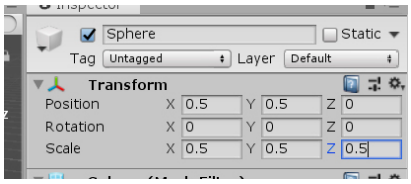
Je to rozhodnutí, takže není žádná **správná** odpověď, rozhodneme se tedy do začátku, že hrdina vidět **bude**. A když náš hrdina bude vidět, vybereme mu ještě nějaký tvar, pro jednoduchost zvolíme krychli (Cube), přidáme do scény a přejmenujeme na **Hrdina**.

Poznámka: *V dalším textu budeme někdy psát o hrdinovi obecně (s malým „h“) a někdy o Hrdinovi jako konkrétním objektu (s velkým „H“), i když ta hranice nemusí být vždycky ostrá, a podobně u dalších objektů. Snad to nebude rušit.*

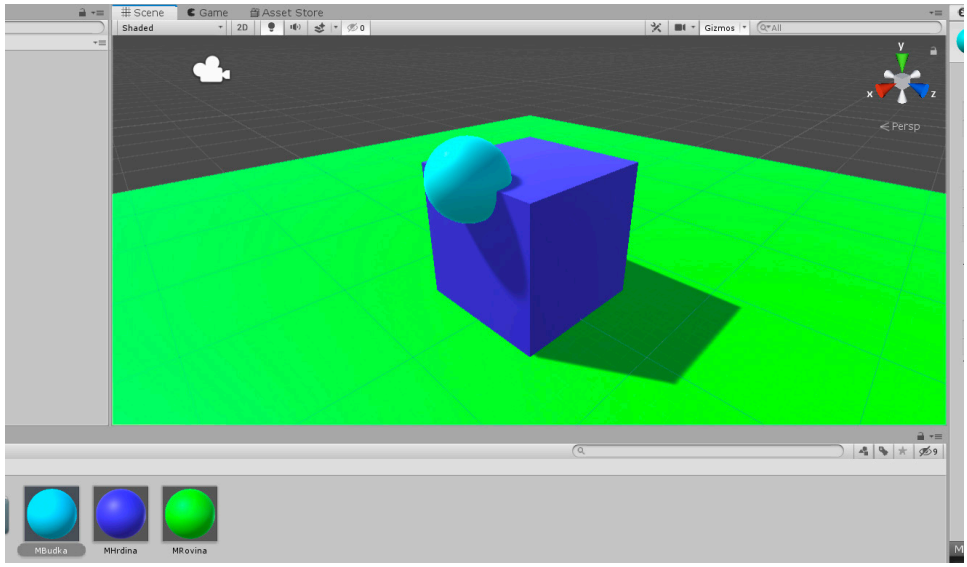
Abychom viděli, jak je krychle-hrdina otočená, tak k ní ještě přidáme kouli (Sphere). Kouli přidáme tak, že přidávací menu vyvoláme na objektu Hrdina, takže nově vytvořený objekt bude součástí hrdiny a budou se ho týkat přesuny, skripty, kolize a další.



Nově vytvořené kouli nastavíme posunutí a velikost (vlastnosti **Position** a **Scale** komponenty **Transform**)



a aby to bylo dobře vidět, tak krychli i kouli přiřadíme nějaký materiál, materiály si pro pořádek pojmenujeme **MHrdina** a **MBudka:**



Když chceme ovládat hrdinu tak se musíme nejdříve něco dozvědět o **skriptech,** protože pohybovat budeme pomocí skriptu, ale také **o ovládání,** neboli o tom, jak číst uživatelovy vstupy. Postupně.

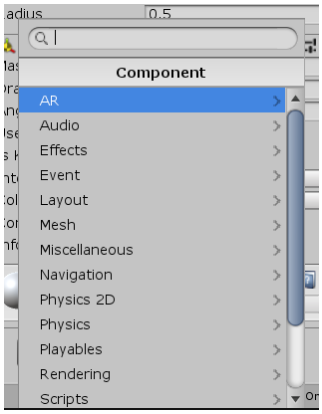
3.4 Skript

Už jsme slyšeli, že se v Unity dá programovat. A už jsme u toho!

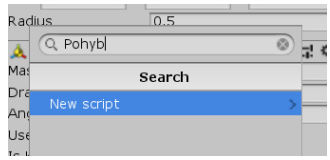
Programovat budeme v jazyku C#, hodí se mít nainstalované Visual Studio nebo jiný nástroj, ve kterém bychom mohli psát a upravovat zdrojové texty.

Skript můžeme vytvořit několika způsoby, buďto pro konkrétní objekt nebo jako jeden z **assets**, podobně jako třeba materiál, který následně přetáhneme/navěsíme na ty objekty, na které budeme chtít.

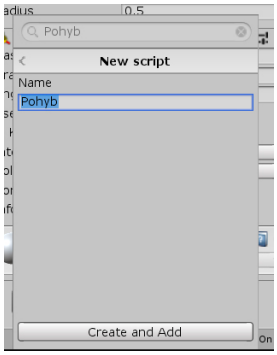
Pro začátek si vybereme první způsob – vytvořit skript jako komponentu pro konkrétní objekt: vybereme si hrdinu, zvolíme **Add Component** a do vyhledávacího pole nahoře



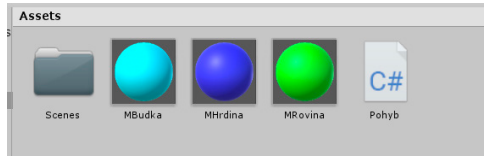
napíšeme název skriptu **Pohyb**. Na to zmizí nabídka a objeví se tlačítko **New script**, které stiskneme:



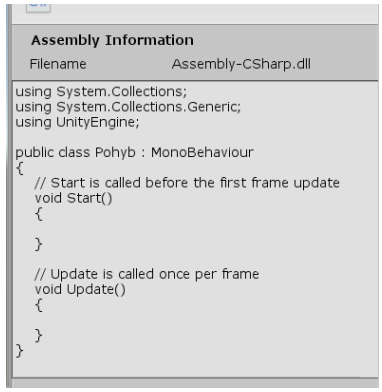
Na to následuje podobný dialog, kde ještě můžeme změnit jméno a potom skript vytvořit a připojit:



Nově vytvořený skript se objeví i v panelu **Assets**:

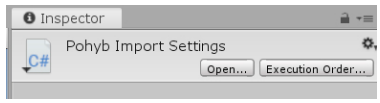


a když ho vybereme, v Inspektoru uvidíme náhled kódu (a protože tento skript je krátký, tak ho vidíme celý):

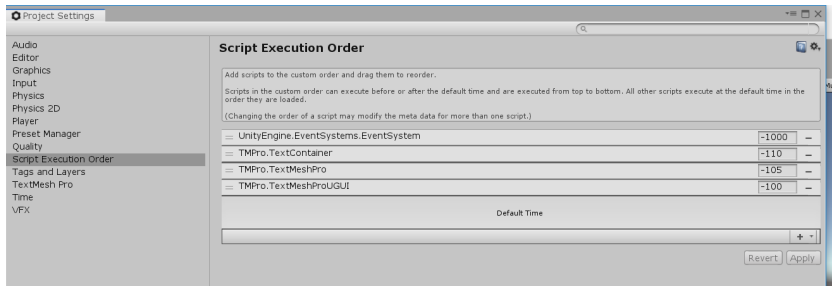


Komentáře vysvětlují, že funkce **Start()** se volá na začátku (to by se nám hodilo pro počáteční vytvoření bludiště a umístění hrdiny a tyranosaura) a funkce **Update()** se volá jednou v každém snímku výsledné hry – zde je tedy příležitost popsat chování postavy, například její reakci na klávesy.

Protože ve výsledném projektu můžeme mít více skriptů a může nám záležet na tom, v jakém pořadí se budou procházet (například jestli se dříve bude provádět skript ovládající hrdinu nebo skript ovládající tyranosaura), můžeme si toto pořadí předepsat kliknutím na tlačítko **Execution order...**



a nastavením pořadí u těch skriptů, kde nás to zajímá:

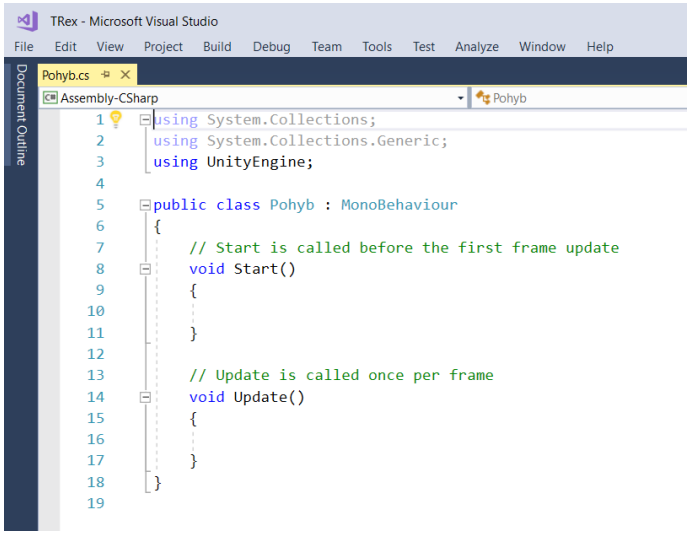


Zatím nás to ale nezajímá, tak nic nenastavujeme.

Takže zpátky, chceme pomocí šipek ovládat pohyb hrdiny a máme skript **Pohyb** a v něm funkci **Start()**, která se zavolá jednou na začátku a funkci **Update()**, která se bude volat v každém snímku.

Ladící výpisy

Pokud se chceme přesvědčit, že se obě funkce opravdu budou volat, můžeme do nich přidat ladící tisky. Na to si skript poklepáním na jeho ikonku otevřeme ve Visual Studiu, na verzi příliš nesejde, teď právě použiji volně dostupné **Microsoft Visual Studio Community 2017**:



Ladící tisk vyvoláme funkcí **Debug.Log()**, kterou budeme volat jak ve funkci **Start()**, tak ve funkci **Update()**:

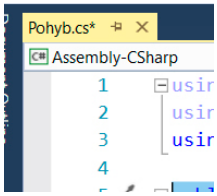
```
public class Pohyb : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("START: " + this.name);
    }

    // Update is called once per frame
```

```
    void Update()
    {
        Debug.Log("UPDATE: " + this.name);
    }
}
```

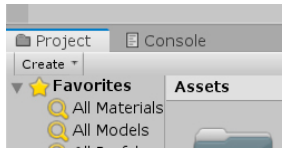
Poznámka: *Když budu v textu zmiňovat nějakou funkci, budu ji psát se závorkami (třeba* ***Update()****), i když to nemusí znamenat, že ta funkce nemá žádné parametry.*

Kromě slova „START“ nebo „UPDATE“ budeme tisknout ještě hodnotu vlastnosti **name** aktuálního objektu.

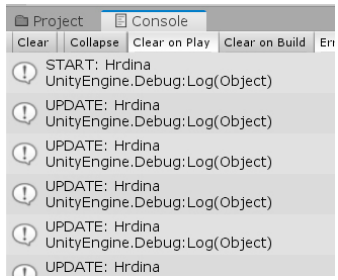
Aby se změny projevily, nezapomeneme skript uložit (neuložený soubor poznáme podle hvězdičky u jména souboru na záložce textového editoru)



a pak se můžeme vrátit do Unity (Visual Studio necháme otevřené) a stiskneme tlačítko **Play** (třeba kombinací kláves **Ctrl-P**). Když po chvíli běh naší hry (ještě nic nedělá) zase ukončíme (třeba stejnou kombinací kláves), přepneme se v **okně projektu** ze záložky **Project** na záložku **Console**



a uvidíme místo, kam se vypisují ladící výpisy



a vidíme, že zpráva „START“ se vypsala jednou na začátku a potom opakovaně zpráva „UPDATE“ , podle toho, jak dlouho jsme nechali hru běžet. Také můžeme vidět, že náš objekt se jmenuje **Hrdina.**

Ladicí tisky jsou cenným nástrojem ve chvíli, kdy něco nefunguje tak, jak má, i když dnes už máme i jiné nástroje, možná se k nim dostaneme.

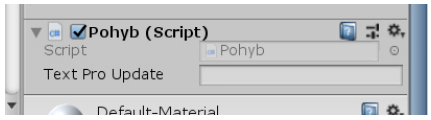
FixedUpdate

Když jsme u toho, tak můžeme také dodat, že GameObject má ještě metodu **FixedUpdate()**, která se volá **pravidelně**. Tato funkce slouží k provádění fyzikálních výpočtů, kde by nepravidelnost vadila, a volá se pravidelně, zatímco četnost volání funkce **Update()** může být vyšší nebo nižší podle složitosti hry, vykreslování apod. Pro běžné účely budeme používat funkci **Update()**.

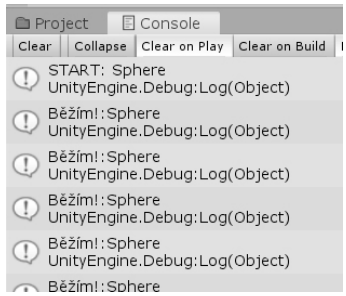
Veřejné proměnné a Inspektor

Odbočka: Dobře, Unity v určitých chvílích volá skript, ale ta vazba může být i opačným směrem. Celý skript popisuje jednu třídu, odvozenou od třídy `MonoBehaviour`, pokud této třídě přidáme nějakou veřejnou vlastnost, třeba `TextProUpdate`, budeme ji moci vidět a nastavovat v Unity v okně **Inspektoru:**

```
public class Pohyb : MonoBehaviour
{
    public string TextProUpdate;
    ...
    // Update is called once per frame
    void Update()
    {
        Debug.Log(TextProUpdate+":" + this.name);
    }
}
```



Když tam potom vyplníme hodnotu, třeba „Běžím!“, nemusíme nijak měnit skript a přitom běžící program bude vypisovat:



Mimochodem, asi už jste si všimli tlačítek, která dovolují vymazat obsah okna Console (**Clear**) nebo nastavit, že se má vymazat při každém spuštění (**Clear on Play**).

Takže ještě neumíme naším objektem pohybovat, ale už umíme propojit objekt se skriptem.

3.5 Pohyb

Když chceme pohybovat herním objektem, ať už na začátku nebo v průběhu hry, použijeme k tomu jeho komponenty.

Komponenty objektu a skript

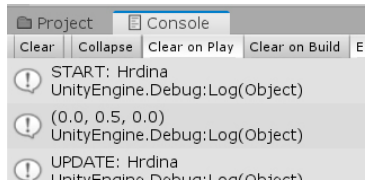
Pokud ze skriptu potřebujeme přístup k nějaké komponentě objektu, získáme ho (generickou) funkcí **GetComponent<>()**, například

```
Rigidbody rb = GetComponent<Rigidbody>();
```

Komponentu **Transform** ale máme přístupnou přímo, takže když do skriptu přidáme výpis

```
Debug.Log(transform.position);
```

vypíše do ladícího okna současnou polohu objektu:



Nás ale více než čtení hodnoty bude zajímat nastavování, protože bychom potřebovali do naší hry přidat pohyb!

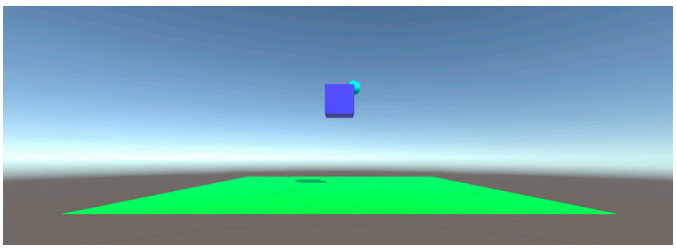
Vector3

Pozice objektu a další hodnoty jsou typu **Vector3** – trojrozměrný vektor se složkami **X**, **Y** a **Z**, tak, jak ho známe z okna Inspektoru. Ukážeme si postupně několik možností, co s ním dělat.

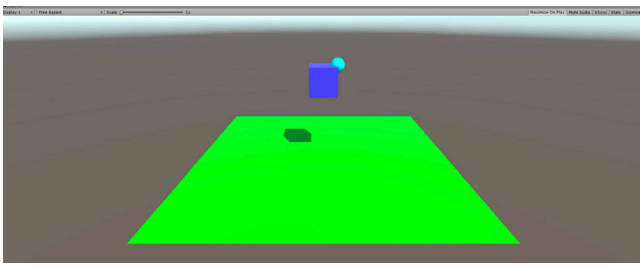
Možnost 1: Pomocí stejnojmenného konstruktoru (programujeme v C#!) můžeme vytvořit nový vektor a dosadit ho:

```
void Start()
{
    transform.position = new Vector3(0, 3, 0);
}
```

Výsledek bude, že se na začátku objekt přesune na nové souřadnice. V nové poloze zůstane po celou dobu, protože nemá komponentu **Physics.Rigidbody** a tedy na něj nepůsobí gravitace:



Mimochodem, měli bychom si posunout kameru, abychom měli lepší výhled – vybereme objekt **Main Camera,** v tu chvíli se objeví malé okénko s pohledem z kamery, abychom nemuseli přepínat – a nastavíme mu posunutí **Position.Y** na **5** (posuneme kameru nahoru) a **Rotation.X** na **30** (nebude se dívat rovně, ale dolů ve směru 30°):



Kdybychom chtěli, můžeme ještě posuvníkem **Scale** změnit velikost (zoom), ale nechme to být.

Takže dokážeme skriptem změnit polohu objektu, pokud ale chceme objektem pohybovat, potřebovali bychom jeho polohu měnit opakovaně a tedy v metodě **Update()**.

Možnost 2: Druhá možnost tedy bude opakovaně měnit hodnotu **transform.position.**

```
void Update()
{
    transform.position += new Vector3(0.1f, 0, 0);
}
```

Pro typ Vector3 jsou přetížené operátory sčítání, přičítání a další, takže nemusíme měnit polohu po složkách a můžeme psát intuitivní kód, včetně operátoru přičítání **+=.** Ta hodnota **0.1f**, tedy jedna desetina jako typ **float** je výsledek experimentů, ale stejně to není dobře, protože nevíme, jak často se bude metoda **Update()** volat a dokonce to ani nemusí být pravidelné, takže když hra zabloudí do nějaké složitější situace, Update se bude volat méně často a pohyb se zpomalí. To není dobře.

Možnost 3: Třetí možnost je zeptat se, kolik času uplynulo od posledního volání funkce Update() a posouvat podle toho:

```
void Update()
{
    transform.position += Time.deltaTime * new Vector3(0.1f, 0, 0);
}
```

Hodnota Time.deltaTime je v sekundách, takže teď by velikost vektoru měla odpovídat tomu, o kolik se má objekt posunout za sekundu.

Možnost 4: Možná už také nějaký čas sledujete, jak se v každém volání funkce Update() vytváří nový objekt typu Vector3 a říkáte si, že je zbytečné takhle zatěžovat správce paměti. Mohli bychom si vektor pohybu vytvořit jen jednou a pamatovat si ho, ale také můžeme použít připravené vektory ve třídě Vector3:

```
void Update()
{
    Debug.Log("UPDATE: " + this.name);
    //transform.position += Time.deltaTime * new Vector3(0.1f, 0, 0);
    transform.position += Time.deltaTime * Vector3.right;
}
```

Kromě **right** třída **Vector3** obsahuje vektory pro dalších pět směrů.

Možnost 5: ...přijde až za chvíli, až se naučíme něco o ovládání.

Ovládání

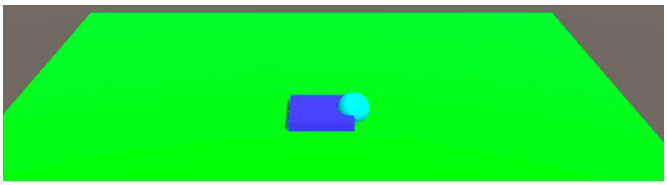
Když umíme objektem pohybovat, hodilo by se pohybovat s ním tam, kam chce uživatel. Na to máme několik funkcí a možností.

Funkce **Input.GetKey()** říká, jestli je stisknutá klávesa, na kterou se ptáme. Takže když funkce **Update()** použije funkci **Input.GetKey()**

```
void Update()
{
    if (Input.GetKey("right"))
        transform.position += Time.deltaTime * Vector3.right;
    if (Input.GetKey("left"))
        transform.position += Time.deltaTime * Vector3.left;
}
```

můžeme objektem pohybovat pomocí šipek.

Kdyby nás napadlo pohybovat se ve směrech **up** a **down**, tak to jde také, ale pozor, u Vektoru3 hodnota **up** znamená opravdu nahoru a **down** opravdu dolů:



a pro pohyb „k sobě“ a „od sebe“ (ve směru osy Z) slouží hodnoty a směry **forward** a **back**:

```
void Update()
{
    if (Input.GetKey("right"))
        transform.position += Time.deltaTime * Vector3.right;
    if (Input.GetKey("left"))
        transform.position += Time.deltaTime * Vector3.left;
    if (Input.GetKey("up"))
```

```
            transform.position += Time.deltaTime * Vector3.forward;
        if (Input.GetKey("down"))
            transform.position += Time.deltaTime * Vector3.back;
    }
```

Relativní pohyb

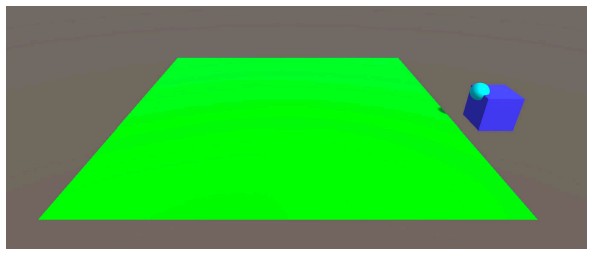
Teď přichází ta výše zmíněná pátá možnost pohybu:

Možnost 5: Pokud budeme chtít chodit s hrdinou v bludišti, je trochu nezvyklé, aby šipka nahoru znamenala vždycky pohyb na sever, šipka doleva pohyb na západ a podobně, spíš je obvyklé, že šipkou nahoru se hrdina pohybuje **kupředu**, ve směru svého pohledu a šipkami doleva a doprava se otáčí.

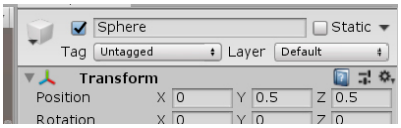
Potřebujeme tedy šipkou nahoru měnit pozici podle aktuálního pootočení a šipkami do stran se pootáčet (**SPEED** je nějaká hodnota typu **float**):

```
    void Update()
    {
        if (Input.GetKey("right"))
            //transform.position += Time.deltaTime * Vector3.right;
            transform.Rotate(0, 90f, 0);

        if (Input.GetKey("left"))
            transform.Rotate(0, -90f, 0);

        if (Input.GetKey("up"))
            transform.position += SPEED * gameObject.transform.forward;
    }
```

Když tento kód spustíme, zjistíme jednak, že jsme si kouli určující směr otočení hrdiny umístili špatně a jednak že máme problém, protože při jednom stisknutí šipky doleva nebo doprava se odehraje více volání funkce Update() a je těžké se otočit právě o 90 stupňů.



První problém vyřešíme tak, že kouli přemístíme:



a druhý tak, že místo funkce **Input.GetKey()** použijeme funkci **Input.GetKeyDown()**, která se liší tím, že pro každé zmáčknutí klávesy vrací **True** jenom jednou:

```
void Update()
{
    if (Input.GetKeyDown("right"))
        transform.Rotate(0, 90f, 0);

    if (Input.GetKeyDown("left"))
        transform.Rotate(0, -90f, 0);

    if (Input.GetKeyDown("up"))
        transform.position += SPEED * gameObject.transform.forward;
}
```

Pohyb s fyzikou neboli Použij sílu!

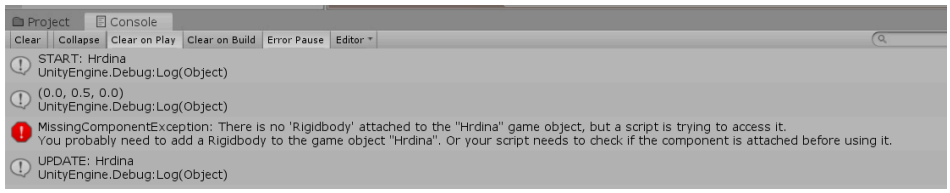
Možnost 6: Uvedený příklad mění polohu hrdiny po skocích a to je to, co potřebujeme, když chceme napodobit starou hru. Na druhou stranu, když máme k dispozici fyziku, nemusíme sami posouvat objektem, ale stačí mu nastavit rychlost nebo ještě lépe zapůsobit na něj silou.

Protože to, co máme teď, budu chtít dál použít, tak jenom přidáme příkaz do metody **Start()** a nakonec ho zase vymažeme.

Protože fyzika působí na komponentu **Rigidbody** daného objektu, vyrobíme si pro ni proměnnou a zapůsobíme na ni silou – buďto opět ve tvaru trojrozměrného vektoru nebo zadáním složek síly ve směru **x**, **y** a **z**:

```
void Start()
{
    Rigidbody rb = GetComponent<Rigidbody>();
    rb.AddForce(-500, 0, 0);
}
```

Pokud se nic neděje, je dobré se podívat do okna konzole a možná tam najdeme zprávu, že dotyčný objekt žádnou komponentu **Rigidbody** nemá, v tom případě mu ji přidáme:



...a můžeme pozorovat, že „nakopnutá“ kostka se odkulí.

Jak velká má být síla, záleží na hmotnosti krychle a koeficientu tření krychle i podložky, ale takto s objekty ve hře zatím pohybovat nebudeme, šlo mi jen o ilustraci toho, jak můžeme sahat na komponenty objektu a tím jej přimět k pohybu.

Umíme tedy

Umíme tedy připojit k objektu skript, využít jeho metodu **Start()** i **Update()**, zjišťovat, jestli **je** nebo **byla právě teď** stisknutá klávesa, číst a měnit pozici a orientaci objektu. Dost na to, abychom mohli začít programovat hru.

4 Hra

4 Hra

4.1 Prefab

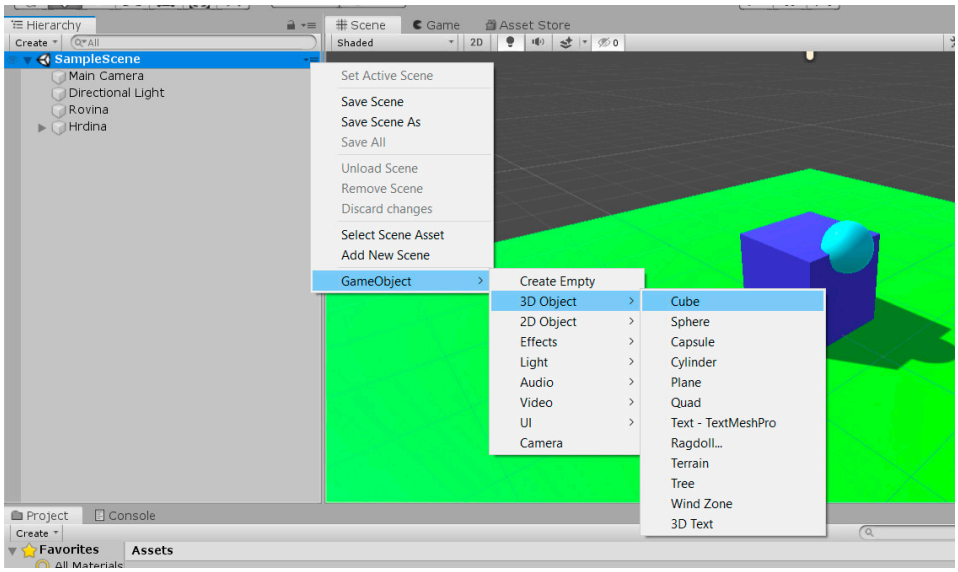
Pokud chceme naprogramovat něco jako zmíněné bludiště s tyranosaurem, potřebujeme přidat nějaké zdi nebo překážky, jinak by to bloudění bylo snadné.

Překážky bychom mohli vytvořit třeba z krychlí nebo jen samotných zdí (zůstaneme u krychlí), ale pokud bychom je prostě vytvořili a rozmístili ve scéně pomocí editoru, bylo by to jednak pracné a jednak by to bludiště vypadalo pokaždé stejně. Budeme je tedy vytvářet a rozmisťovat pomocí skriptu a na to se nám hodí něco, čemu se v Unity říká **prefab** (český název neznám a ani bych ho nehledal, zůstaňme u názvu „prefab“).

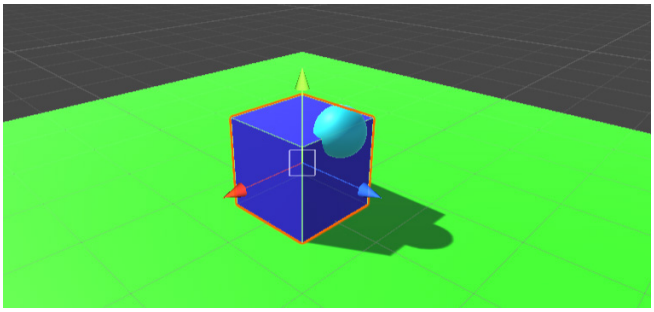
Prefab slouží jako vzor, podle kterého můžeme pomocí skriptů vytvářet **instance.** Výhoda oproti kopii je v tom, že pokud změníme prefab, změní se všechny instance – a také v tom, jak už zmíněno, že instance můžeme vyrábět skriptem, hodí se to třeba na vytváření střel ve hrách, ve kterých se střílí nebo čehokoliv, co nebude ve scéně od začátku.

Vytvoření prefab-u

Prefab vytvoříme tak, že nějaký objekt přetáhneme z okna Hierarchie do okna Project (nejlépe do složky Assets). Takže si ve scéně vytvoříme krychli

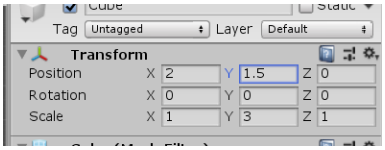


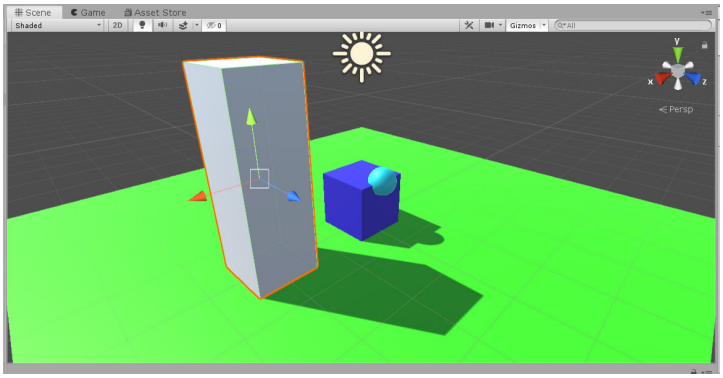
Pokud ji nevidíme, je to proto, že byla vytvořená na stejném místě jako náš objekt Hrdina:



– tak ji trochu posuneme:

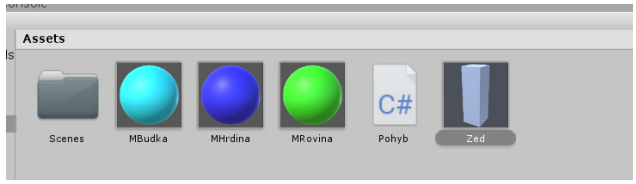
Mohli bychom ji také trochu natáhnout do výšky, aby Hrdina nekoukal přes zeď – a posunutí i zvětšení výšky můžeme místo tahání myší udělat nastavením vlastností **Transform**: posuneme ji změnou **Position.X**, zvětšíme změnou **Scale.Y**, ale protože zvětšením výšky se zeď zanoří pod povrch, vynoříme ji změnou **Position.Y:**



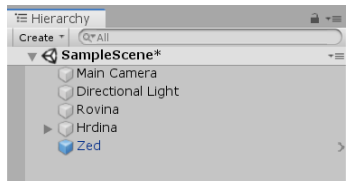


Jako poslední akci si nový prvek pojmenujeme (v okně Hierarchy, F2), zvolíme jméno **Zed** (bez háčků).

Abychom teď objekt **Zed** změnili na prefab, chytíme ho myší v okně Hierarchy a přetáhneme mezi Assets:



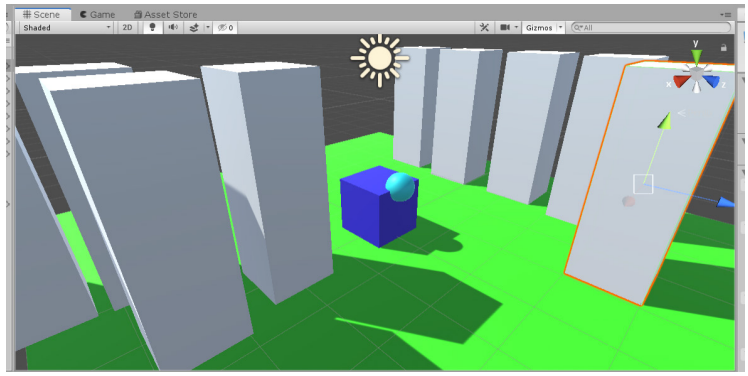
V okně Hierarchy má teď **Zed** změněnou ikonku (to záleží na nastavení Unity):



A pokud bychom teď naši hru spustili, uvidíme jedinou instanci, kterou jsme ve scéně zanechali.

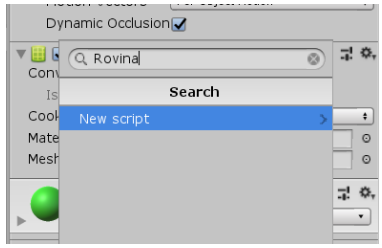
Instance prefabu

Pokud chceme v editoru scény přidat do scény instance nějakého prefabu (vím, že to skloňování anglických slov není hezké), jednoduše je chytíme v okně Assets a přetáhneme:



Zajímavější ovšem bude výroba instancí prefabu pomocí skriptu, proto tyto instance zase smažeme.

Budeme chtít, aby skript vyrábějící zdi patřil k objektu **Rovina**, tak vybereme objekt **Rovina**, v Inspektoru klikneme na **Add Component** a místo vybírání z nabídky napíšeme do vyhledávacího pole název skriptu **Rovina**



a klikneme na **New script** a pak ještě jenou na tlačítko **Create and add**.

Nový skript **Rovina** (soubor se jmenuje Rovina.cs, ale Unity názvy skriptů zobrazuje bez přípony, tak je budu psát také tak) otevřeme poklepáním a protože instance objektů budeme chtít vytvářet jenom jednou při spuštění hry, budeme měnit metodu **Start().**

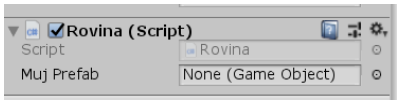
Skript ale potřebuje odkaz na prefab, ze kterého chceme vytvářet instance, proto ve skriptu vytvoříme veřejnou proměnnou, ta bude tím pádem viditelná v Inspektoru a tam vybereme příslušný prefab, tedy postupně:

1) do skriptu přidáme veřejnou proměnnou pro prefab:

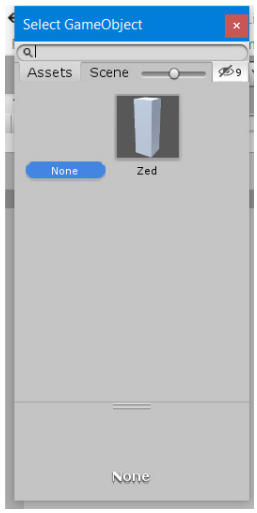
```
public class Rovina : MonoBehaviour
{
    public GameObject mujPrefab;

    // Start is called before the first frame update
    void Start()
    {

    }
```

...a nezapomene skript uložit.

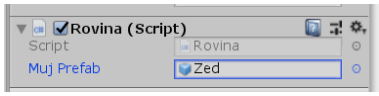
2) tato veřejná proměnná bude viditelná v okně Inspektoru (pořád ještě máme vybraný objekt **Rovina**),



a u políčka **MujPrefab** klikneme na ikonku napravo od hodnoty a ze zobrazené nabídky (přepněte se na záložku **Assets**)



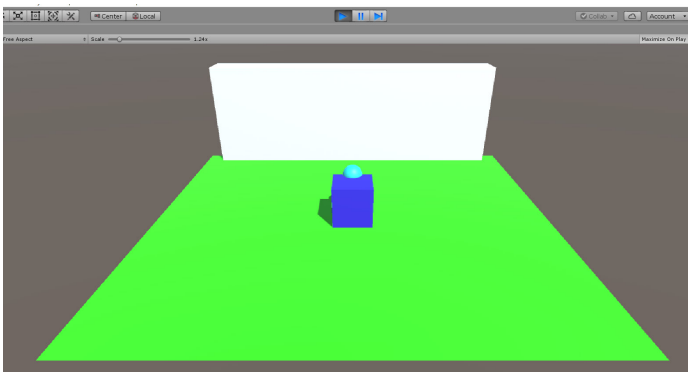
vybereme prefab **Zed**:



Samotné vytvoření instance potom obstará funkce **Instantiate()**, která má jako parametry prefab, pozici a otočení. Když už tu funkci píšeme, tak si těch zdí vyrobíme více:

```
void Start()
{
    for (int x = -4; x < +5; x++)
    {
        Instantiate(mujPrefab, new Vector3(x, 1.5f, 5), Quaternion.identity);
    }
}
```

Souřadnice **X** nově vytvořených objektů se mění od -4 do +4, souřadnice **Y** odpovídá objektu stojícímu na Rovině a souřadnice **Z** je nejvzdálenější umístění nad **Rovinou**. Pootočení **Quaternion.identity** udává žádné pootočení:



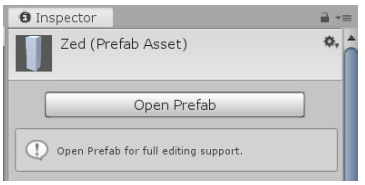
Poznámka: *Pokud zkusíme šipkami pohnout naším hrdinou proti zdi a pokud budeme vytrvalí, zjistíme, že umí procházet zdí. Ukázali jsme si několik možností, jak pohybovat objektem a ta s nastavováním polohy není ideální v kombinaci s překážkami a fyzikou.*

4.2 Jak změnit prefab

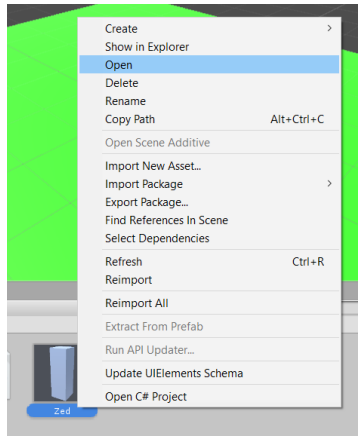
Pokud změníme prefab, projeví se změna ve všech instancích.

Abychom prefab mohli změnit, potřebujeme ho otevřít v režimu prefabu (**prefab mode**). Pokud nějakou instanci prefabu máme uvnitř scény, stačí na ni v okně Hierarchie poklepat nebo kliknout na šipku napravo od instance.

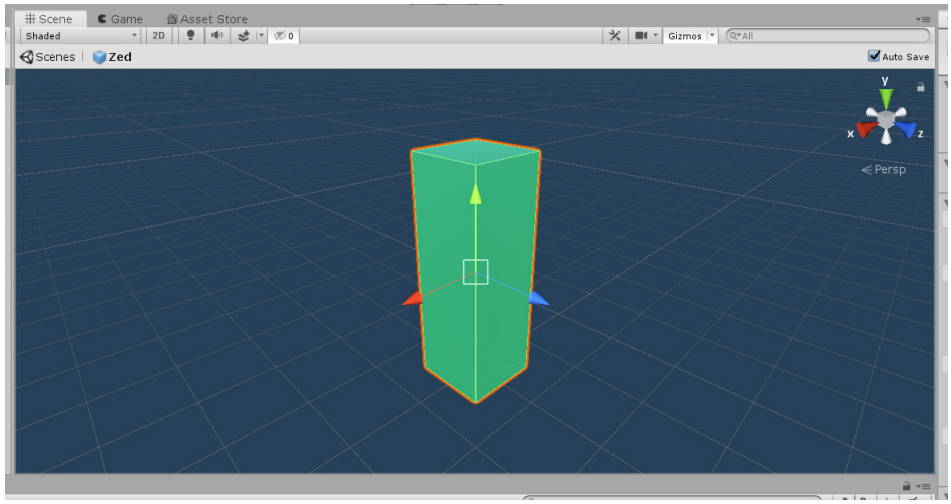
Pokud žádnou instanci prefabu ve scéně nemáme, můžeme vybrat prefab v seznamu **Assets** a potom v okně Inspektoru kliknout na tlačítko **Open Prefab**



Nebo také můžeme v seznamu **Assets** na prefabu pravým tlačítkem vyvolat lokální menu a vybrat příkaz **Open**

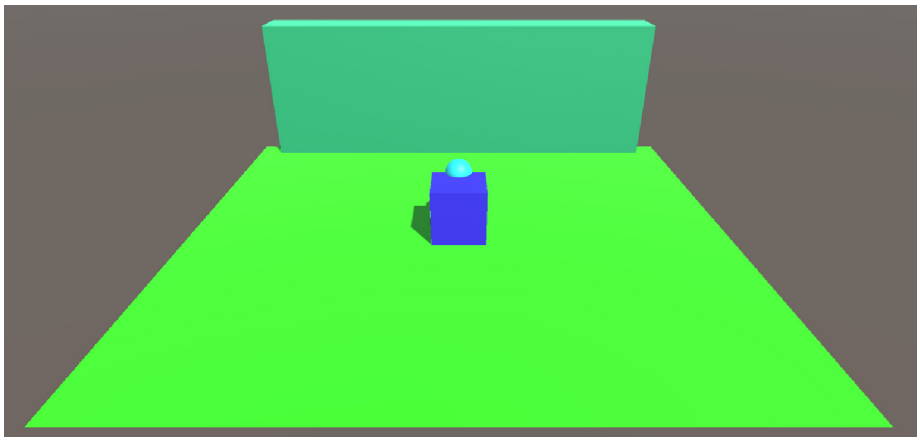


Tak či onak se ocitneme v režimu, kdy můžeme prefab upravovat, nastavovat mu vlastnosti, přidávat komponenty a třeba mu nastavit materiál přetažením materiálu **MZed** (právě jsme si ho bez vysvětlování vyrobili) ze seznamu Assets na zobrazený prefab:



Zpátky se vrátíme kliknutím na položku **Scenes** v záhlaví editoru nebo na šipku doleva v okně hierarchie:

Pokud program spustíme, všechny instance prefabu **Zed** mají nastavený materiál **MZed:**



Dovedeme tedy vytvořit prefab, umíme vytvářet jeho instance a umíme prefab i dodatečně změnit tak, že se změna projeví ve všech jeho instancích.

4.3 Pohyb kamery

Kdybychom chtěli naprogramovat nějaké bloudění v bludišti (naše bludiště je zatím hodně jednoduché), potřebovali bychom, aby hráč neměl takhle celé bludiště jako na dlani, ale aby viděl jen to, co vidí hrdina. Budeme potřebovat pohybovat kamerou.

Protože budeme chtít pohybovat kamerou pokaždé, když se pohne hráč, budeme s ní hýbat ve skriptu **Pohyb** a protože s ní budeme chtít hýbat v každém kroku, bude to ve funkci **Update()**.

Pohledy na hru

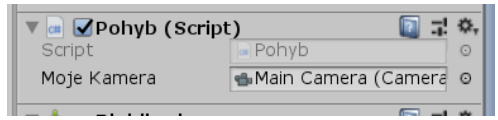
Ve hrách se používá jak pohled první osoby, tedy očima hrdiny, tak pohled třetí osoby, kdy je vidět i postava hrdiny. Při pohledu první osoby stojí kamera na místě hrdiny a tedy hrdina není vidět (a takového máme krásného hrdinu!), při pohledu třetí osoby stojí kousek za hrdinou, aby kromě samotného pohledu hrdiny byl vidět i hrdina samotný. Začněme pohledem první osoby, protože to bude o trochu jednodušší.

Pohled první osoby

Skript **Pohyb** je přiřazen objektu **Hrdina**, má tedy přístup k vlastnostem tohoto objektu, ale pokud chceme, aby mohl manipulovat kamerou, musíme mu ji předat jako parametr. Jak na to, to už jsme viděli, přidáme proměnnou:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Pohyb : MonoBehaviour
{
    public Camera mojeKamera;
```

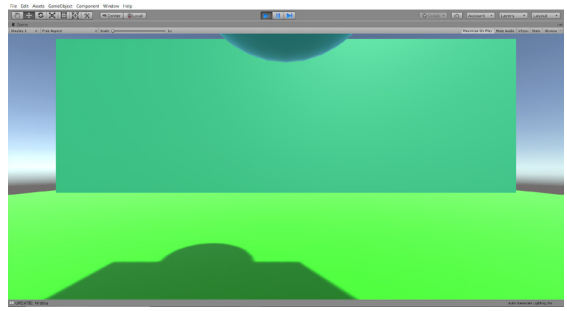
...a v okně Inspektoru do ní přetáhneme kameru – objekt **Main Camera**.



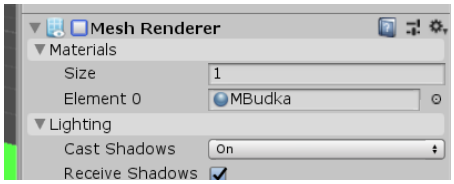
Funkce **Update()** obsahuje testování stisknutých kláves a podle toho úpravu polohy a orientace hrdiny – to necháme a jen na konec funkce přidáme dva řádky, které nastaví pozici a orientaci kamery podle pozice a orientace hrdiny:

```
        mojeKamera.transform.position = transform.position;
        mojeKamera.transform.rotation = transform.rotation;
    }
```

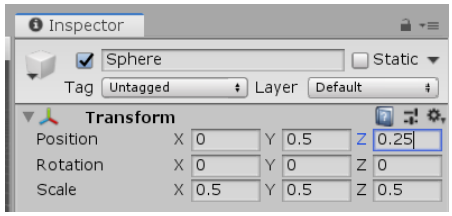
Když hru spustíme teď, najednou ji uvidíme z pohledu hrdiny:



Ta ošklivá věc nahoře je ta koule („budka“), kterou jsme přidali abychom viděli orientaci hrdiny, protože kamera je umístěná ve středu hrdiny a dívá se ven. Můžeme se jí zbavit tak, že ji odstraníme (i když… hrdina není vidět, ale je vidět jeho stín a to je docela pěkné) nebo tak, že jí odškrtneme komponentu **Mesh Renderer**



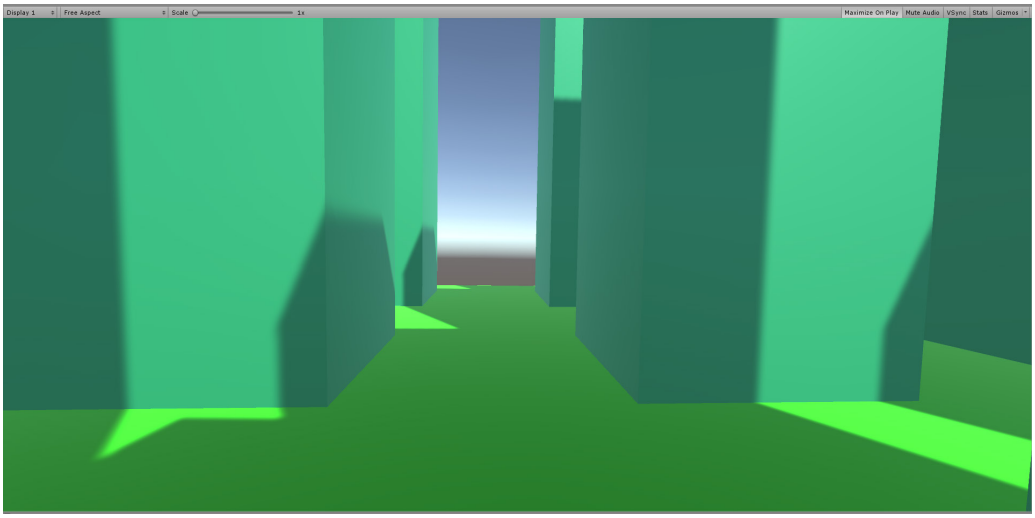
, ale to také zmizí stín nebo ji posuneme tak, aby se dostala mimo pohled kamery (a tím se zároveň mohou vyřešit problémy s jejím narážením do okolí):



Teď už přichází chvíle na to, abychom si připravili trochu složitější scénu, protože ji vyrábíme skriptem (skriptem **Rovina**), tak to bude jen změna zdrojového kódu:

```
void Start()
{
    for (int x = -5; x <= +5; x++)
    {
        for (int z = -5; z <= +5; z++)
        {
            Debug.Log($"{x}:{z}");
            if ((x%2==0)&&(z%2==0))
                Instantiate(mujPrefab, new Vector3(x, 1.5f, z), Quaternion.identity);
        }
    }
}
```

...a výsledný pohled vypadá takto:

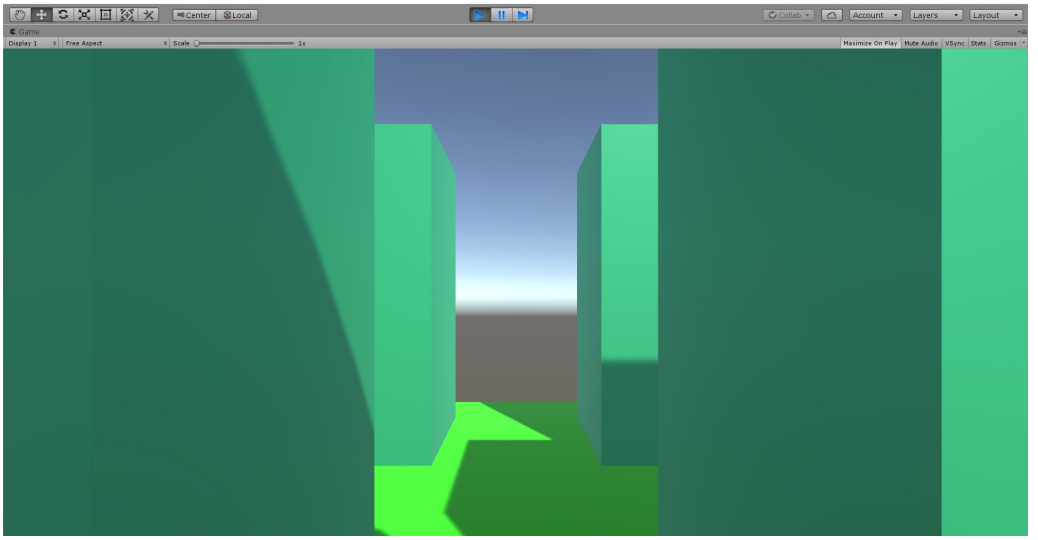


V bludišti s tyranosaurem byla vidět i horní část zdí, na to můžeme buďto zvednout kameru vzhůru nebo zmenšit výšku zdí. Zmenšit výšku zdí můžeme buďto změnou prefabu nebo tím, že nově vytvořeným instancím nastavíme vlastnost **transform.localScale**. Ale když budeme výsledným instancím zmenšovat výšku, musíme také změnit jejich umístění, aby se nevznášely ve vzduchu.

Výsledná funkce potom bude vypadat takto:

```
void Start()
{
    for (int x = -5; x <= +5; x++)
    {
        for (int z = -5; z <= +5; z++)
        {
            if ((x % 2 == 0) && (z % 2 == 0))
                Instantiate(mujPrefab, new Vector3(x, 1, z), Quaternion.identity)
                    .transform.localScale = new Vector3(1, 2, 1);
        }
    }
}
```

Výchozí pozice hrdiny je ale **(0, 1, 0)**, to znamená, že stojí v řadě zdí, přidejme tedy do funkce **Start()** ještě příkaz, který ho posune na křižovatku:



To už připomíná původní hru (zkuste si s hrdinou šipkami pohnout).

Pohled třetí osoby

U pohledu třetí osoby máme několik možností. První z nich by byla prostě kameru připojit k objektu hrdiny, pokud se nám to nelíbí (nelíbí), můžeme to udělat skriptem a oproti pohledu první osoby potřebujeme jen kamerou trochu couvnout (od pozice odečíst směr pohledu, případně pronásobený nějakým číslem) a když nám bude vadit, že z celé hry vidíme jen záda hrdiny, tak kameru ještě trochu zvednout:

```
//mojeKamera.transform.position = transform.position;
//mojeKamera.transform.rotation = transform.rotation;

mojeKamera.transform.position += transform.up;
mojeKamera.transform.position -= 3*transform.forward;
```

Nevýhoda takového pohledu třetí osoby je v tom, že někdy prostě „tři kroky za hrdinou“ není to správné místo pro kameru – třeba proto, že je tam zeď.

Jiná možnost, jak řešit pohled třetí osoby, je mít kameru na pevném místě, která bude jen měnit orientaci tak, aby byl hrdina v centru záběru. To ale zase není příliš šikovné v bludišti, kde takto hrdinu nejspíš neuvidíme. Další možnosti mají zase další problémy, takže zůstaneme u pohledu první osoby.

4.4 Znovu pohyb hrdiny

Návrh

Viděli jsme, že máme několik způsobů, jak pohybovat postavou hrdiny – změnit polohu skokem při stisknutí klávesy, měnit polohu postupně, pokud je stisknutá klávesa, nebo nastavit rychlost, případně zapůsobit silou a nechat pohyb na simulaci fyziky obsažené v Unity. Podobné spektrum možností máme, pokud jde o otáčení. Problém je ale v tom, že když se má hrdina pohybovat v bludišti nad pravoúhlou čtverečkovou sítí, potřebuje se otáčet o devadesát stupňů a posunovat se tak, aby skončil na správném místě, protože jinak se mu nepodaří zahýbat do vedlejších chodeb, buďto proto, že nemá správný směr nebo proto, že nestojí přímo proti odbočující chodbě.

Mohli bychom to trochu ošidit tím, že hrdinu zmenšíme a také tím, že změníme jeho tvar z krychle na válec nebo kouli, protože krychle se v chodbě špatně otáčí, mohli bychom i trochu podvádět a pozici i orientaci v případě potřeby dorovnat na středy kostek a směry rovnoběžné s osami… je to, jak to někdy v programování bývá – můžeme naprogramovat, co budeme chtít, ale potřebujeme si rozmyslet, co tedy budeme chtít.

Zkusme proto teď (a to nezáleží na Unity) navrhnout, jak by mělo fungovat ovládání hrdiny, aby to bylo hezké a příjemné pro hráče.

Chtěli bychom aby pohyb i otáčení probíhaly postupně a ne skokem, aby to vypadalo hezky.

Zároveň bychom potřebovali, aby pohyb i otáčení končily na těch správných místech a úhlech.

Proto navrhujeme toto:

Otočení:
Po stisku otáčecí klávesy se hrdina začne plynule (aby to bylo hezké na pohled) otáčet a otáčení skončí, až bude otočený jedním ze čtyř směrů rovnoběžných s osami X a Z (aby byl správně otočený pro další pohyb).

Pohyb:
Po stisku klávesy pohybu se hrdina začne pohybovat vpřed, pohyb skončí, až bude stát na další pozici (ve středu čtverce).

Máme návrh, můžeme ho naprogramovat a zkusit, jak se nám to jako hráčům bude líbit.

Pokud se vám nelíbí, navrhněte si vlastní pravidla ovládání.

Naprogramovat

Několik poznámek k tomu, jak navržená pravidla naprogramovat.

Pohyb

Když hráč stiskne klávesu pro pohyb, zahájíme pohyb. Protože reagujeme na stisknutí a ne na držení, budeme si někde pamatovat, že probíhá pohyb – a pokud probíhá pohyb, budeme se posouvat o díl odpovídající uplynulému času **Time.deltaTime**.

To pamatování – mohli bychom mít proměnnou, která říká, zda právě teď probíhá pohyb, nebo, protože něco podobného budeme potřebovat i pro otáčení, bychom mohli mít jedinou proměnnou, udávající, co právě děláme. Protože začínáme pohybem a nechceme to zatím komplikovat, zůstaneme u proměnné pro pohyb. A protože potřebujeme, aby držela informaci mezi různými voláními funkce **Update()**, nebude deklarovaná uvnitř této funkce, ale bude to členská proměnná třídy:

```
bool probihaPohyb = false;
```

Když probíhá pohyb, posuneme hrdinu o patřičný díl cesty v patřičném směru, ale potřebujeme u toho hlídat, jestli v tomto kroku nepřekročí cílové políčko. To bychom dokázali s trochou počítání a také řešení pro čtyři různé směry, ale můžeme si pomoci trikem: když se pohyb bude rozbíhat, tak si spočítáme a uložíme cílové souřadnice (to je snadné, doteď jsme se na ně přesouvali skokem) a zapamatujeme si, **za jak dlouho na ně dojedeme**. A během jednotlivých kroků místo toho, abychom kontrolovali souřadnice nebo úhel, budeme kontrolovat jenom to, **zda už uběhl celkový čas**. Vtipné, ne?

Rychlost a rozměry

Když přičítáme k poloze hrdiny vektory jako **transform.forward**, přičítáme (nebo odečítáme) jedničku k dané souřadnici. Když jsme vytvářeli instance prefabu **Zed**, umisťovali jsme je na celočíselné souřadnice. Když se ptáme na uplynulý čas pomocí **Time.deltaTime**, dostáváme hodnotu v sekundách. Takže pokud nastavíme cílovou pozici na současnou pozici plus směr, posouváním po **Time.deltaTime** bychom tam dojeli za sekundu. Když se budeme chtít pohybovat vyšší rychlostí, budeme muset touto rychlostí násobit změnu souřadnice a dělit zbývající čas. Uděláme si na to proměnnou a pokud tato proměnná bude zase členskou proměnnou objektu, dovolí nám to v průběhu hry měnit rychlost pohybu hrdiny:

```
float rychlostPohybu = 5;
```

Otáčení

S otáčením to bude podobné, cílovou orientaci si uložíme jako trojici úhlů

```
Vector3 cilovaOrientace;
```

a požadovanou hodnotu získáme tak, že současnou orientaci hrdiny převedeme na úhly a připočteme k nim násobek vektoru **(0, 1, 0)**, tedy rotace podle svislé osy Y:

```
cilovaOrientace = transform.rotation.eulerAngles + smerOtaceni * Vector3.up;
```

Pořadí

Musíme ještě ohlídat správné pořadí, ve kterém se ptáme na jednotlivé podmínky, tedy jestli probíhá pohyb, jestli probíhá otáčení – a teprve potom jestli je stisknutá nějaká klávesa. To nám zamíchá řešení pohybu s řešením otáčení, ale není vyhnutí.

Zkušenost

Když to vyzkoušíme, zjistíme, že je trochu otravné pro každý krok znovu mačkat klávesu, tak změníme volání funkce **Input.GetKeyDown()** na volání funkce **Input.GetKey()**.

A ještě jeden postřeh – pokud v tomto kroku skončí pohyb nebo otáčení, na nastartování dalšího pohybu musíme čekat až do příštího snímku, proto ještě změníme pořadí. Hodilo by se nám v případě nekončícího pohybu nebo otáčení z funkce vyskočit příkazem **return** a jinak se ptát na stisknutou klávesu, ale to bychom přišli o pohyb kamery na konci funkce. Příkaz skoku používat nechceme, tak si pomůžeme ošklivým trikem – cyklus, ze kterého budeme vyskakovat příkazem **break**:

```
bool probihaPohyb = false;
Vector3 cilovaPozice;
float zbyvajiciCas;
float rychlostPohybu = 5;

bool probihaOtaceni = false;
float smerOtaceni;
Vector3 cilovaOrientace;
float rychlostOtaceni = 5;

void Update()
{
    while (true)
```

```
{
    if (probihaPohyb)
    {
        zbyvajiciCas -= Time.deltaTime;
        if (zbyvajiciCas <= 0)
        {
            transform.position = cilovaPozice;
            probihaPohyb = false;
        }
        else
        {
            transform.position +=
                 Time.deltaTime * rychlostPohybu * transform.forward;
            break;
        }
    }

    if (probihaOtaceni)
    {
        zbyvajiciCas -= Time.deltaTime;
        if (zbyvajiciCas <= 0)
        {
            transform.eulerAngles = cilovaOrientace;
            probihaOtaceni = false;
        }
        else
        {
            transform.Rotate(0, Time.deltaTime * rychlostOtaceni * smerOtaceni, 0);
            break;
        }
    }

    if (Input.GetKey("up"))
    {
        cilovaPozice = transform.position + transform.forward;
        probihaPohyb = true;
        zbyvajiciCas = 1f / rychlostPohybu;
    }
    else
    if (Input.GetKey("left") || Input.GetKey("right"))
    {
```

```
                if (Input.GetKey("left"))
                    smerOtaceni = -90;
                else
                    smerOtaceni = 90;

                cilovaOrientace = transform.rotation.eulerAngles + smerOtaceni * Vector3.
up;
                probihaOtaceni = true;
                zbyvajiciCas = 1f / rychlostOtaceni;
            }
            break;
        }

        mojeKamera.transform.position = transform.position;
        mojeKamera.transform.rotation = transform.rotation;
    }
```

Jde to naprogramovat ještě jinak

Naprogramování pohybu, jak jsme ho viděli výše, vycházelo z toho, že celkový pohyb (nebo i otáčení, ale zůstaňme u pohybu) potřebujeme rozložit mezi jednotlivá volání funkce **Update()**. V Unity ovšem máme ještě jednu možnost a to použít **korutinu** – podprogram, který bude přerušován voláním **yield return**:

```
    protected IEnumerator PlynulyPohyb(Vector3 konec)
    {
        float zbyvajiciVzdalenostsqr = (transform.position - konec).sqrMagnitude;

        while (zbyvajiciVzdalenostsqr > float.Epsilon)
        {
            Vector3 novaPozice
              = Vector3.MoveTowards(rb.position, konec, rychlostPohybu * Time.deltaTime);
            rb.MovePosition(novaPozice);
            zbyvajiciVzdalenostsqr = (transform.position - konec).sqrMagnitude;
            yield return null;
        }
    }
```

Pohyb potom zahájíme voláním

```
                StartCoroutine(PlynulyPohyb(transform.position + transform.forward));
```

Příklad tohoto pohybu můžeme najít v Unity-tutoriálu **2D Roguelike** (https://learn.unity.com/project/2d-roguelike-tutorial).

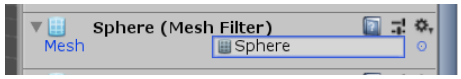
Změna pozice a fyzika

Oba předvedené postupy mají jednu nevýhodu, totiž, že měníme pozici objektu a to jde špatně dohromady s fyzikou. Když totiž fyzika ví, že se blíží objekt, který má určitou rychlost, může zareagovat a objekt nepustit, případně ho odrazit a podobně, podle nastavení fyzikálního materiálu. Ale když nějakému objektu řekneme „od teďka budeš stát tady!“, tak je pro fyziku těžké reagovat a výsledky jsou někdy nepředvídatelné. Proto bychom změnu pozice měli používat pouze tehdy, když jsme si zjistili, že na dané pozici nic není – a to si ukážeme později.

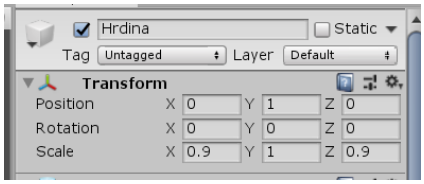
4.5 Tvar hrdiny

Už jsme se zmiňovali o tom, že hrdina ve tvaru krychle může mít problémy s otáčením v chodbách bludiště a že by bylo šikovnější, kdyby měl jiný tvar. I když nám to teď nevadí, protože se otáčíme pouze na křižovatkách, ukažme si změnu tvaru hrdiny.

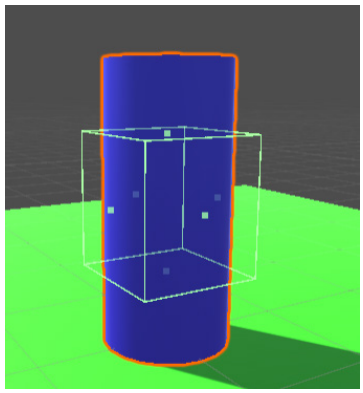
První krok, který ale nestačí, je změnit tvar Hrdiny na válec (šla by i koule):



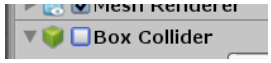
a také vymažeme jeho část – kouli, kterou jsme používali pro označení směru – a ještě Hrdinu trochu zmenšíme:



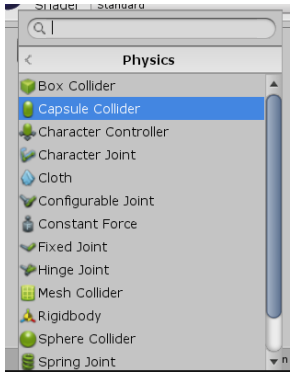
Kdybychom se teď chtěli s Hrdinou otáčet uprostřed chodeb, tak to stále nepůjde a proč, to uvidíme, když si zobrazíme pozměněného Hrdinu v editoru scény:



Hrdina má totiž pořád ještě komponentu **Box Collider**, která se stará o kolize s okolím – a která má tvar hranolu! Box Collider tedy smažeme nebo jenom vypneme:



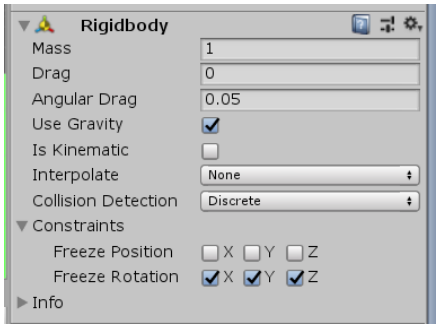
a přidáme Hrdinovi novou komponentu – **Capsule Collider**:



Teď už by se hrdina mohl otáčet i uprostřed chodeb, kdyby to potřeboval.

Neklopit!

A ještě jedna možnost, která by se nám mohla někdy hodit: Kdybychom hrdinou (ve tvaru vysokého válce) naráželi do jiných objektů, mohl by se nám překlopit. Můžeme to zakázat v jeho vlastnostech:



Teď už nám nespadne ani kdyby do něj narazil Tyranosaurus Rex.

5 Bludiště a další

5 Bludiště a další

5.1 Bludiště

Bludiště, ve kterém má bloudit náš hrdina, je zatím trochu jednoduché. Tak se pojďme podívat na způsoby, jak vytvořit složitější bludiště, a na věci, které s tím souvisí.

Vytvoření bludiště, i když zatím hodně jednoduchého, řeší skript **Rovina.cs**, necháme to v něm i nadále. To, že bludiště je tvořeno instancemi prefabu **Zed**, necháme také a zkusíme se jen podívat na to, jak vyrobit složitější a větší bludiště.

Velikost

Zatím jsme se moc nestarali o to jak jsou naše objekty veliké, nanejvýš jsme je zvětšovali nebo zmenšovali pomocí vlastnosti **Transform.Scale**, ale teď to budeme potřebovat vědět. Nebo naopak, pokud budeme chtít vyrobit bludiště o daném rozměru, budeme muset patřičně zvětšit nebo zmenšit objekt **Rovina**, který tvoří jeho podlahu.

Z objektu **Rovina** se podíváme na jeho komponentu typu **Collider** (Rovina má ve skutečnosti odvozený typ **Mesh Collider**, ale to nám nevadí), získáme ho funkcí **GetComponent()**:

```
Collider collider = GetComponent<Collider>();
```

Když máme collider, můžeme zjistit jeho velikost pomocí jeho vlastnosti **bounds.size**, ale na výslednou velikost má ještě vliv nastavené škálování, takže když chceme znát skutečnou velikost Roviny (zatím jsme ji nijak neměnili), spočítáme ji jako

```
Debug.Log($"velikost: {Vector3.Scale(transform.localScale, collider.bounds.size)}");
```

(tu funkci **transform.Scale()** používáme k tomu, abychom nemuseli velikosti v jednotlivých osách násobit každou zvlášť):

```
Debug.Log($"bounds.size: {collider.bounds.size}");
Debug.Log($"scale: {transform.localScale}");
Debug.Log($"výsledná velikost: "
    +$"{Vector3.Scale(transform.localScale, collider.bounds.size)}");
```

a výsledek je:

```
bounds.size: (10.0, 0.0, 10.0)
scale: (1.0, 1.0, 1.0)
```

```
výsledná velikost: (10.0, 0.0, 10.0)
```

Rovina má tedy výchozí velikost 10 krát 10 a budeme ji škálovat podle velikosti požadovaného bludiště.

Kde vzít bludiště

Myslíme tím otázku „jak se dozvědět, kde mají být Zdi a kde volno" – a možností je víc. Můžeme ho například mít pevně zadané v programu nebo ho můžeme načítat ze souboru (jak se z Unity dostat k souborům?) nebo ho můžeme generovat náhodně (a jak zařídit, aby bylo hezké a průchozí?).

Abychom oddělili tvorbu bludiště od přípravy scény, navrhneme si **formát**, ve kterém budou bludiště uchovávána a předávána, a potom budeme moci mít zvlášť jednak funkce, které budou bludiště vyrábět a jednak část funkce **Start()**, která podle daného bludiště připraví scénu.

Tento formát pro representaci informace o bludišti by mohl být dvourozměrné pole znaků (na **zeď** nebo **volno** by stačila logická hodnota, ale časem možná budeme mít v tomto poli víc informací), nebo ještě lépe pole textových řetězců (to můžeme také procházet dvěma indexy, ale bude lépe vidět jeho obsah).

Metoda **Start()** bude toto pole zpracovávat, nastaví velikost roviny a umístí Zdi (a později možná ještě něco dalšího):

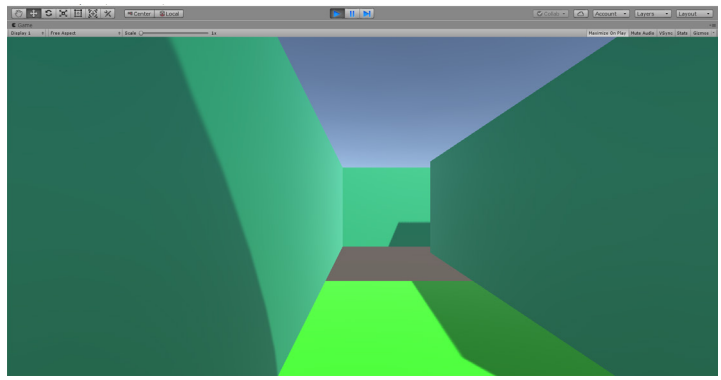
```
const char ZED = ‚X‘;
void Start()
{
    // ziskat bludiste:
    string[] blud = Bludiste_A();

    // zjistit rozmery:
    int sx = blud[0].Length;
    int sz = blud.Length;

    // prizpusobit velikost Roviny:
    transform.localScale = new Vector3(sx / 10, 1, sz / 10);

    // nasazet Zdi:
    for (int x = 0; x < sx; x++)
    {
        for (int z = 0; z < sz; z++)
        {
```

```
            if (blud[z][x]== ZED)
                Instantiate(mujPrefab, new Vector3(x, 1, z), Quaternion.identity)
                    .transform.localScale = new Vector3(1, 2, 1);
        }
    }
    return;
}
```

Funkce **Bludiste_A()** jenom vrátí připravené pole textových řetězců:

```
string[] BLUDISTE_A =
{
        "XXXXXXXXXX",
        "X        X",
        "X  XXXXXX",
        "X  X     X",
        "X  XXX XXX",
        "X  X   X X",
        "X  X  X  X",
        "X  XXXX X",
        "X        X",
        "XXXXXXXXXX"
};

string[] Bludiste_A()
{
    return BLUDISTE_A;
}
```

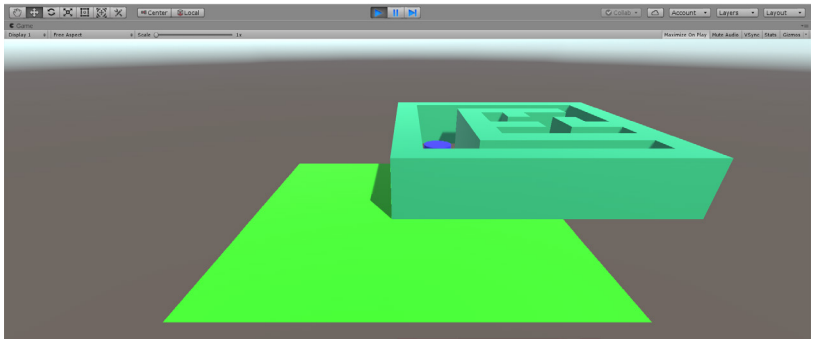
Pohled hráče vypadá takto



a bylo by to dobré, až na tu divnou podlahu!

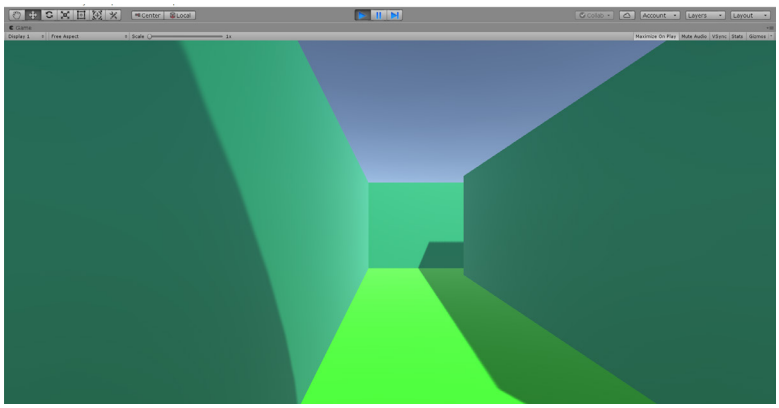
Pokud chceme zjistit, v čem je problém, pomůže nám zakomentovat ve skriptu **Pohyb.cs** řádky, kterými posouváme kameru do pohledu hráče:

```
//mojeKamera.transform.position = transform.position;
//mojeKamera.transform.rotation = transform.rotation;
```



– a vidíme, že objekt **Rovina** je chybně umístěný. Jeho atribut **transform.position** totiž stejně jako u jiných objektů udává souřadnice **středu** a ne nějakého rohu. Do metody **Start()** proto ke změně velikosti podle velikosti bludiště přidáme ještě posun:

```
// prizpusobit velikost Roviny - a posunout ji:
transform.localScale = new Vector3(sx / 10, 1, sz / 10);
transform.position += new Vector3((sx-1) / 2f, 0, (sz-1) / 2f);
```



5.2 Pozice hrdiny

Máme tedy jednoduchý způsob, jak popsat bludiště (zatím ještě neřešíme jeho načítání ani generování), ale už asi vidíme, že tam schází informace o umístění a směru pohledu hrdiny.

Domluvme se, že hrdina bude v poli bludiště znázorněn jedním ze znaků `^ > v <`, podle toho, kterým směrem se bude dívat, upravme ukázkový popis BLUDISTE_A a jeho zpracování ve funkci **Start()**.

```
string[] BLUDISTE_A =
{
        "XXXXXXXXXX",
        "X       <X",
        "X  XXXXXXX",
        "X  X     X",
        "X  XXX XXX",
        "X  X   X X",
        "X  X X   X",
        "X  XXXXX X",
        "X        X",
        "XXXXXXXXXX"
};
```

Skript **Rovina** bude potřebovat přístup k objektu **Hrdina**, mohli bychom mu pro něj zase vyrobit veřejnou proměnnou a v okně Inspektoru do ní přetáhnout nebo vybrat objekt Hrdina, ale protože objekt Hrdina budeme potřebovat jenom jednou, můžeme si ho nechat vyhledat:

```
GameObject hrdina = GameObject.Find("Hrdina");
```

Možností, jak vyhledat objekt, je více, zde jsme použili vyhledání podle jména (pozor při přejmenování!).

Protože teď už v bludišti potřebujeme rozpoznávat pět různých znaků (jeden pro zeď a čtyři pro hrdinu), nahradíme podmíněné příkazy **if** příkazem **switch**.

```
        // nasazet Zdi a umistit a nasmerovat Hrdinu:
        GameObject hrdina = GameObject.Find("Hrdina");
        for (int x = 0; x < sx; x++)
        {
            for (int z = 0; z < sz; z++)
            {
```

```
            switch (blud[z][x])
            {
                case ZED:
                    Instantiate(mujPrefab, new Vector3(x, 1, z), Quaternion.identity)
                        .transform.localScale = new Vector3(1, 2, 1);
                    break;
                case '^':
                    hrdina.transform.position = new Vector3(x, 1, z);
                    hrdina.transform.eulerAngles = 0*Vector3.up;
                    Debug.Log("^");
                    break;
                case '<':
                    hrdina.transform.position = new Vector3(x, 1, z);
                    hrdina.transform.eulerAngles = -90 * Vector3.up;
                    Debug.Log("");
                    break;
                case '>':
                    hrdina.transform.position = new Vector3(x, 1, z);
                    hrdina.transform.eulerAngles = 90 * Vector3.up;
                    break;
                case 'v':
                    hrdina.transform.position = new Vector3(x, 1, z);
                    hrdina.transform.eulerAngles = 180 * Vector3.up;
                    break;
            }
        }
    }
```

Poznámka: *Může být matoucí, že řádky v popisu bludiště jsou indexovány od nuly odshora k vyšším indexům níže, ale ve scéně je nultá souřadnice* ***Z*** *nejblíže a vyšší hodnoty jsou dále, tedy výše. Takže skutečné bludiště bude proti svému popisu vertikálně překlopené.*

Můžeme to vyřešit třeba změnou řádku

```
switch (blud[z][x])
```

na

```
switch (blud[sz-1-z][x])
```

Poznámka: *Pokud chceme nastavovat pozici a orientaci hrdiny ve skriptu* ***Rovina****, měli bychom zrušit jeho nastavování ve skriptu* ***Pohyb****.*

5.3 Bludiště jako asset

Zatím jsme měli popis bludiště uložený ve zdrojovém kódu programu.

Kdybychom chtěli bludiště načítat ze souboru, potřebovali bychom vědět, kam takový soubor umístit a jak se k němu dostat ze zdrojového kódu, ale aplikace v Unity nepracuje s oddělenými soubory, místo toho používá jeden druh Asset-ů – **TextAsset**.

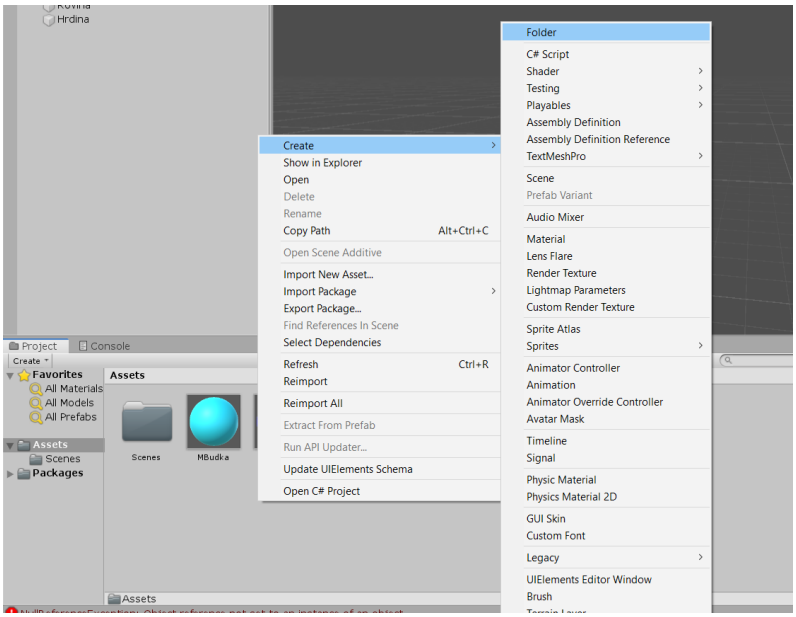
TextAsset získáme tak, že soubor s příponou **.txt**, **.xml** a několik dalších přetáhneme do okna Assets. Na to ho Unity zkonvertuje do svého interního formátu a zpřístupní nám ho pro skripty.

Soubor s bludištěm

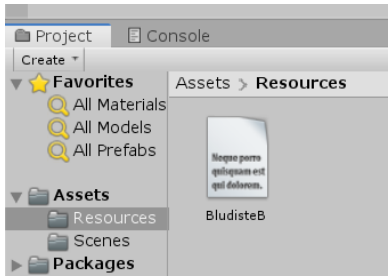
Vytvoříme si tedy nejdříve soubor **BludisteB.txt**:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x                                    x
x  xxxxxxxxx    xxx    xxxxxxxxx   x
x  x            x x           x   x
x  xxxxxxxxx    x x    xxxxxxxxx   x
x               x x                x
x               x x                x
x               x x                x
x      ^                           x
x    xxxxxxxxxxxxxxxxxxxx          x
x                                  x
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Uložený soubor ale nestačí přetáhnout jen do okna **Assets**, potřebujeme si v něm vytvořit složku **Resources**:



a teprve do ní přetáhnout soubor s bludištěm:



Ve skriptu (skript **Rovina**, funkce **Start()**) se na něj potom můžeme odkázat takto:

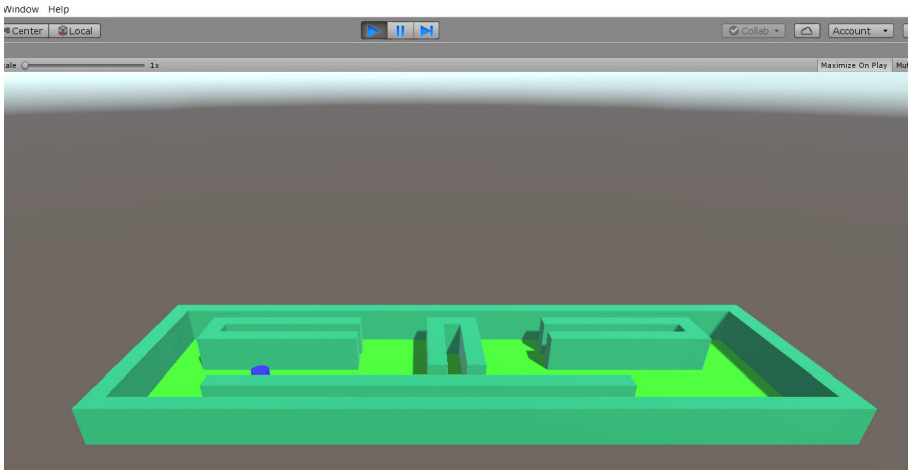
```
string[] Bludiste_B()
{
    TextAsset ta = Resources.Load("BludisteB") as TextAsset;
    char[] odd = { ,\r', ,\n' };
    string[] blud = ta.text.Split(odd, System.StringSplitOptions.RemoveEmptyEntries);
    return blud;
}
```

Objekt **TextAsset** má vlastnost **text**, na rozdělení použijeme metodu **Split()** a pokud si nejsme jisti, jak jsou odděleny řádky (CRLF, CR, LF), použijeme na dělení oba oddělovače, ale parametrem **RemoveEmptyEntries** řekneme, že prázdné položky (případný text mezi CR a LF) se mají vynechat. Šlo by to celé zkrátit do méně příkazů, ale šlo mi o přehlednost.

Pokud zase zakomentujeme pohyb kamery a případně si kameru ve funkci **Start()** trochu posuneme

```
mojeKamera.transform.position = new Vector3(15, 15, -20);
```

bude to vypadat takhle:



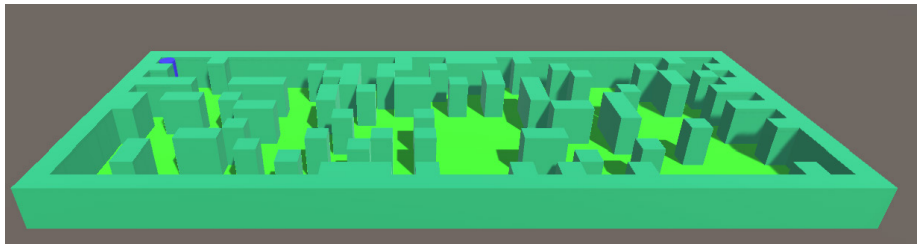
Takže umíme do projektu přidávat externí soubory a z projektu k nim přistupovat (i když tam ve skutečnosti žádné oddělené soubory nebudou).

5.4 Jak generovat bludiště skriptem

To je zajímavá úloha, obzvlášť pokud máme požadavky na to, aby se dalo dojít odkudkoliv kamkoliv, a ještě navíc třeba na to, jak mají být široké chodby – a nesouvisí to s Unity, tak to řešit nebudeme.

Kdybychom chtěli jen náhodné bludiště bez dalších požadavků, tak potřebujeme jen umístit zdi kolem dokola a náhodně vygenerovat pozice zdí a pak nějaké umístění a orientaci hrdiny, třeba takhle:

```
string[] Bludiste_N()
{
    int sx = Random.Range(15, 50);
    int sz = Random.Range(15, 50);
    string[] blud = new string[sz];

    float hustota = 0.20f;
    bool uzBylHrdina = false;

    for (int z = 0; z < sz; z++)
    {
        blud[z] = "";
        for (int x = 0; x < sx; x++)
        {
            if ((x == 0) || (x == sx-1) || (z == 0) || (z == sz-1)
                        || (Random.value < hustota))
                blud[z] += ZED;
            else
            if (uzBylHrdina == false)
            {
                blud[z] += "^";
                uzBylHrdina = true;
            }
            else
                blud[z] += " ";
        }
    }
    return blud;
}
```

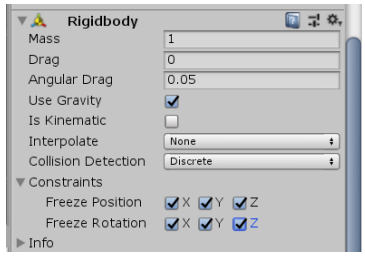


...ale není na tom nic zajímavého.

5.5 Ještě jednou pohyb

Hrdina nám chodí hezky, otáčí se do pravého úhlu a posouvá se na správná místa – ale zkusili jste s ním narazit do zdi? Pokud ano, zjistili jste, že dokáže projít zdí!

Mohlo by nás napadnout, že je to proto, že Zed (prefab) nemá komponentu **Physics.Rigidbody**. Když mu ji přidáme, zjistíme, že hrdina už zdí projít nedokáže – ale když do zdi narazí, tak se Zeď posune! Mohli bychom to řešit „jako ve skutečnosti" tak, že bychom v této komponentě nastavili parametr **Mass**, tedy hmotnost, na nějakou velikou hodnotu. Anebo tak, a to použijeme, že zafixujeme orientaci i pozici:



Teď zeď stojí pevná jako skála – ale hrdina do ní zase znovu může vejít! Musíme to řešit jinak.

5.6 Vyslat paprsek

Funkcí **Physics.Raycast()** můžeme vyslat paprsek z daného místa daným směrem do dané maximální vzdálenosti – a zeptat se, jestli do něčeho narazil, případně do čeho. A pokud před hrdinou je nějaká překážka, zatím se neptáme jaká, tak pohyb vůbec nezahájit:

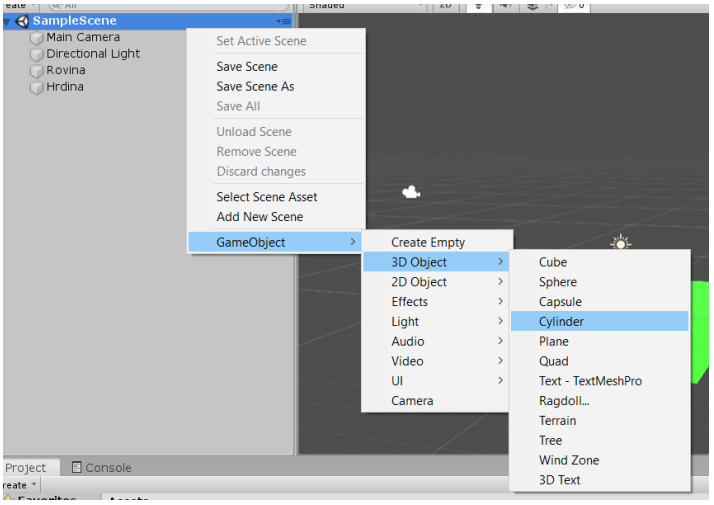
```
if (Input.GetKey("up"))
{
    RaycastHit hitInfo;
    if (!Physics.Raycast(transform.position, transform.forward, out hitInfo, 1f))
    {
        cilovaPozice = transform.position + transform.forward;
        probihaPohyb = true;
        zbyvajiciCas = 1f / rychlostPohybu;
    }
    else
        Debug.Log("nejdu - prekazka");
}
```

Funkce **Physics.Raycast** má mnoho různých způsobů volání a ten výstupní parametr **hitInfo** bychom teď vůbec nepotřebovali, ale bude se nám hodit za chvíli, tak si ho necháme.

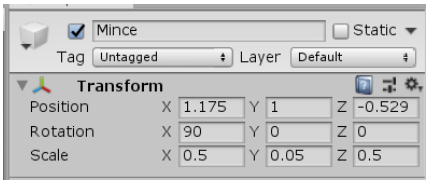
5.7 Sběr

Co kdyby náš hrdina měl v bludišti něco sbírat? Ať už nějaké mince nebo puntíky, které potřebuje vysbírat všechny jako Hungry Horace nebo Pacman, nebo nějaké bonusy, které mu budou poskytovat další schopnosti?

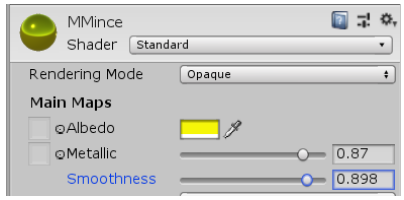
Vyrobit je bude snadné, potřebujeme si jen v Unity vyrobit nový prefab, na to nejdříve přidáme do scény válec (**Cylinder**),



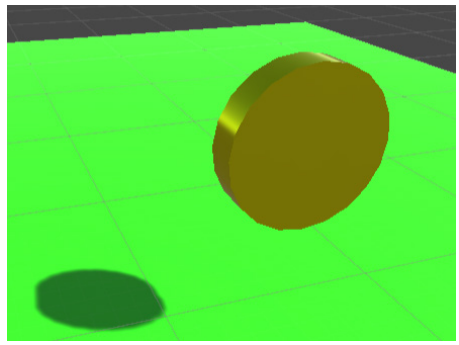
pojmenujeme ho **Mince**, posuneme ho, abychom na něj viděli (ale nakonec ho zase vrátíme na **(0, 0, 0)**), změníme mu velikost (pootočení změníme teprve při vytváření instance):



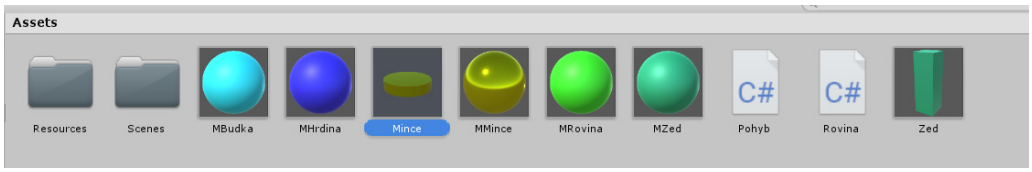
vyrobíme materiál **MMince** a nastavíme mu žlutou barvu, kovový vzhled a hladkost:



a přetáhneme ho na objekt **Mince**:



Nakonec vynulujeme posunutí a objekt **Mince** přetáhneme z okna **Hierarchie** do okna **Assets**, čímž z něj vyrobíme prefab:



Dále si potřebujeme vymyslet, jakým znakem budeme minci znázorňovat v popisu bludiště, třeba písmenem **m** – a upravit popis bludiště...

```
string[] BLUDISTE_M =
{
        "XXXXXXXXXX",
        "X m m m <X",
        "X  XXXXXXX",
        "X  X     X",
        "X  XXX XXX",
```

```
            "X  XmmmX X",
            "X  X XmmmX",
            "X  XXXXXmX",
            "X    mmmmX",
            "XXXXXXXXXX"
    };
```

...a nakonec potřebujeme při vytváření bludiště brát v potaz i znak **m** a na jeho místě vytvářet instanci prefabu **Mince** podobně, jako vytváříme instanci prefabu **Zed**, když vidíme znak **ZED** (taky bychom si na to „m“ měli udělat proměnnou):

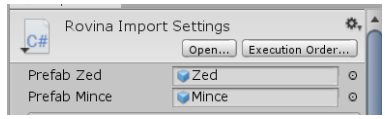
```
        case ZED:
            Instantiate(PrefabZed, new Vector3(x, 1, z), Quaternion.identity)
                .transform.localScale = new Vector3(1, 2, 1);
            break;
        case "m":
            Instantiate(PrefabMince, new Vector3(x, 1.2f, z), Quaternion.Euler(90, 45, 0));
            break;
```

Parametr Quaternion.Euler(90, 45, 0) říká, jak má být instance pootočená. Z hlediska správy programu by bylo lepší správně pootočený prefab Mince umístit do jiného prefabu, z něhož bychom potom vytvářeli instance bez pootočení, ale to teď řešit nebudeme.

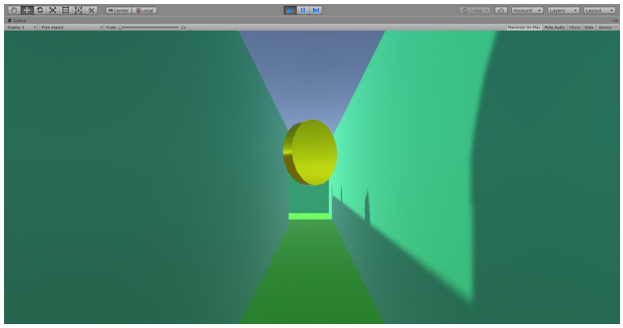
Aby skript **Rovina.cs** mohl používat prefab **Mince**, musíme si na něj zase vytvořit veřejnou proměnnou

```
    public GameObject PrefabZed;
    public GameObject PrefabMince;
```

(původní proměnou **mujPrefab** jsme přejmenovali na vhodnější **PrefabZed**) a v okně Inspektoru skriptu **Rovina** nastavit prefab **Mince** jako hodnotu vlastnosti **prefabMince**:



Výsledek potom vypadá takto:



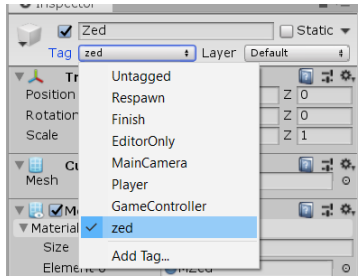
Z dalších mincí v řadě jsou vidět jenom stíny.

Poznámka: *Není úplně snadné najít správnou velikost a umístění mincí, pokud je dáme příliš nízko, neuvidíme minci, kterou máme pod nohama. Pokud ji dáme příliš vysoko, nezaznamená ji náš průzkum paprskem (funkcí* ***Raycast()****) a když do ní při pohybu narazíme, fyzika může tvořit zajímavé efekty.*

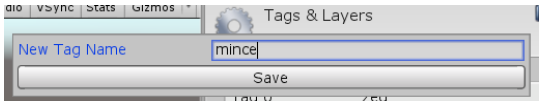
Co tedy ten slíbený sběr

Dobře, máme v bludišti mince, vidíme je, ale nemůžeme procházet, protože pokud Hrdina paprskem zjistí, že je před ním překážka, nezačne se pohybovat. Zmiňovali jsme se o tom, že se můžeme zeptat, co je to za překážku – a teď na to přišel správný čas.

Vlastnost **collider.gameObject** výstupního parametru funkce **Raycast()** vrací objekt, do kterého paprsek narazil a tohoto objektu se pak můžeme zeptat na jméno (objekty, které vytváříme při vytváření bludiště, se jmenují **Zed(Clone)** nebo **Mince(Clone)**, nebo lépe se můžeme zeptat na obsah vlastnosti **tag** a do ní si poznamenat, co potřebujeme: Vybereme si prefab **Zed**, v Inspektoru rozbalíme nabídku **Tag**, pomocí **Add tag** do seznamu přidáme hodnotu **zed** a tu si pak vybereme:



A když už jsme u přidávání značek, rovnou si nadefinujeme i tag **mince** a úplně stejně ho přidáme prefabu **Mince**:



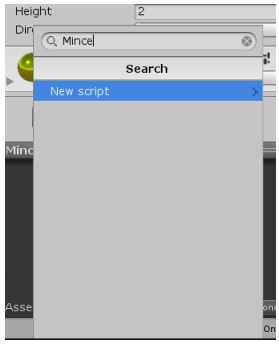
Když se teď v programu zeptáme na vlastnost **tag** objektu, který byl zasažen paprskem, dostaneme hodnotu **zed** a podle toho můžeme upravit rozhodování skriptu **Pohyb** při stisku klávesu kupředu:

```
            if (Input.GetKey("up"))
            {
                RaycastHit hitInfo;

                if (Physics.Raycast(transform.position, transform.forward, out hitInfo,
1f))
                {
                    GameObject predeMnou = hitInfo.collider.gameObject;
                    if (predeMnou.tag=="zed")
                    {
                        Debug.Log("nejdu - prekazka");
                        break;
                    }
                    // else - mince => sbirat...
                }

                // kdyz jsem dosel sem, tak JDU a uz se na nic neptam:
                cilovaPozice = transform.position + transform.forward;
                probihaPohyb = true;
                zbyvajiciCas = 1f / rychlostPohybu;
            }
```

Máme bludiště, máme v něm mince, umíme je při pohybu odlišit od zdí, zeď nám brání v pohybu – a co mince? Záleží na našem rozhodnutí. Mohlo by to vypadat třeba takhle:

Když chceme jít kupředu
    (to chceme, jinak bychom nevysílali paprsek a nezkoumali, co je před námi)
a zjistíme, že před námi je **Mince**, tak řekneme té Minci, ať se zničí
    (na to bude mít Mince nějakou funkci a bude to chvíli trvat)
a až se Mince zničí, tak ohlásí Hrdinovi, že je zničeno
a Hrdina si přičte body a nějak je zobrazí.

Takže vyrobíme skript **Mince** pro minci, třeba prostřednictvím přidání komponenty v okně Inspektoru:



Volání funkce z cizího skriptu

Abychom mohli zavolat funkci, která je definovaná v jiném skriptu, potřebujeme nejdříve najít ten skript:

```
if (predeMnou.tag == "mince")
{
    Mince skriptMince = GameObject.FindObjectOfType(typeof(Mince)) as Mince;
    skriptMince.Seber(predeMnou);
    break;
}
```

Protože tím najdeme skript a funkci, ale ten skript a tu funkci sdílí všechny mince, budeme jako parametr předávat, **která mince** se má zničit.

Poznámka: *Možností, jak vyhledávat objekt a jak popsat, co chceme hledat, je víc; časem uvidíme další.*

Celý skript pro mince potom vypadá takto:

```
public class Mince : MonoBehaviour
{
    float casDoKonceSbirani;
    GameObject sbiranaMince;
```

```
    // Start is called before the first frame update
    void Start()
    {
        casDoKonceSbirani = 0f;
    }

    public void Seber(GameObject mince)
    {
        sbiranaMince = mince;
        casDoKonceSbirani = 1f;
    }

    // Update is called once per frame
    void Update()
    {
        if (casDoKonceSbirani > 0)
        {
            sbiranaMince.transform.eulerAngles += Time.deltaTime * 720 * Vector3.up;
            casDoKonceSbirani -= Time.deltaTime;
            if (casDoKonceSbirani <= 0)
            {
                sbiranaMince.SetActive(false);

                Pohyb skriptPohyb = GameObject.FindObjectOfType(typeof(Pohyb)) as Pohyb;
                skriptPohyb.MinceSebrana();
            }
        }

    }
}
```

Funkce **Seber(GameObject mince)** nastaví, která mince se má sebrat a zbývající čas do konce. Funkce **Update()** potom, pokud čas do konce ještě nevypršel, otáčí (tou sbíranou) mincí, zmenšuje zbývající čas a v okamžiku, kdy čas vyprší, deaktivuje sbíranou minci a pošle zprávu skriptu **Pohyb** pomocí zavolání funkce **MinceSebrana()**.

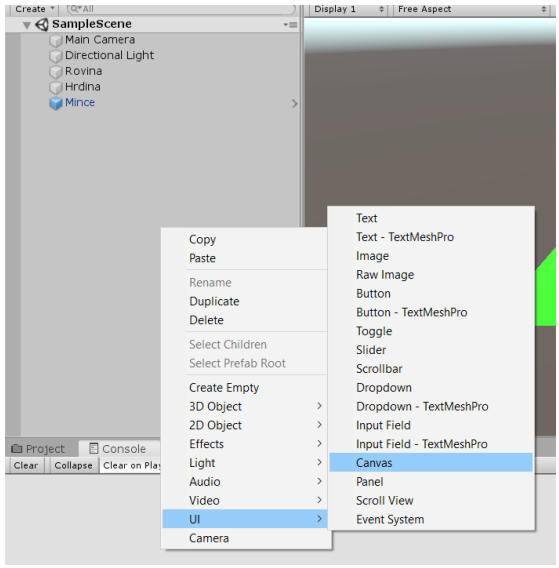
5.8 Počet sebraných mincí a UI

Ve skriptu **Pohyb()** si budeme pamatovat počet sebraných mincí, ale bylo by pěkné, kdybychom to mohli nějak ukázat uživateli. K tomu slouží prvky **uživatelského rozhraní** neboli **UI**.

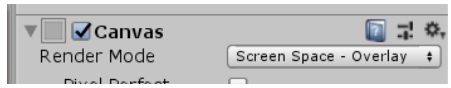
Pro vytváření uživatelského rozhraní existuje v Unity několik knihoven, v současné době to jsou **UIElements**, **Unity UI** a **IMGUI** (ale pravděpodobně bude existovat i řada dalších). Pro naši hru použijeme knihovnu **Unity UI**.

Unity UI

Všechny ovládací prvky musí být uvnitř objektu **Canvas**. Vytvoříme ho v okně hierarchie, stejně jako další ovládací prvky:



V editoru ani po přepnutí na záložku **Game** nic neuvidíme, ale v okně hierarchie můžeme objekt Canvas vybrat a v okně Inspektoru mu nastavovat vlastnosti. Vlastnost **Render Mode** určuje, jakým způsobem se bude Canvas vykreslovat, ponecháme výchozí hodnotu **Screen Space - Overlay** neboli bude se kreslit až na výsledný obrázek a v tom případě nemusíme řešit velikost, protože bude sahat od kraje do kraje:

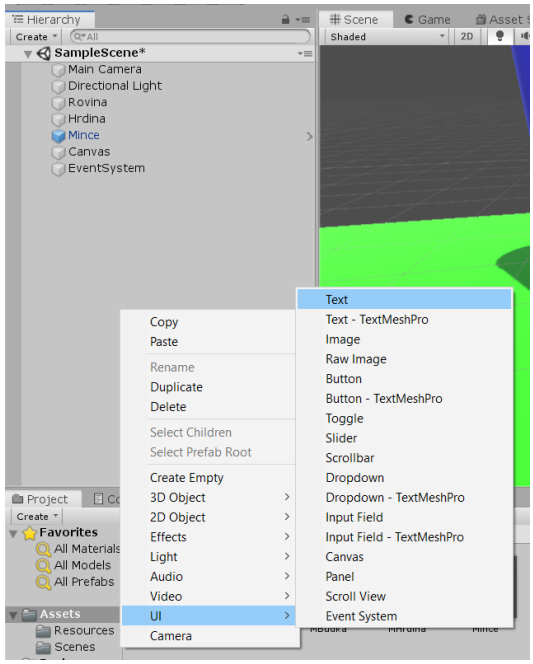


Pro vytvoření textu pole máme (jako u ostatních úkolů) více možností.

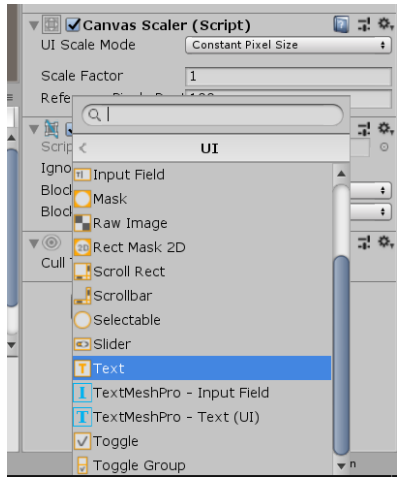
První možnost je vytvořit text jako nový objekt, druhá jako komponentu objektu Canvas.

Text jako nový objekt

Objekt **Text** vytvoříme v okně hierarchie:



a nový objekt se automaticky zařadí pod objekt **Canvas** (předchozí krok jsme mohli přeskočit, vytvořit rovnou objekt **Text** a pokud by žádný objekt **Canvas** neexistoval, vytvořil by se automaticky).

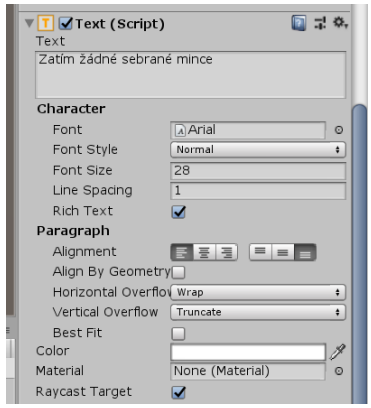


Text jako komponenta objektu Canvas

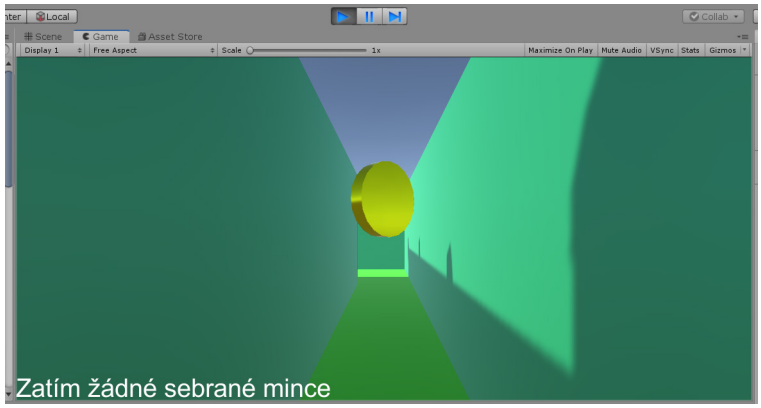
Druhá možnost spočívá v tom, že objekt Canvas vybereme (pokud není vybraný), a v okně Inspektoru mu přidáme komponentu **Text** pomocí **Add Component** a **UI** a **Text**:

Další pokračování je podobné, i když umístění výsledného textu může být pokaždé jiné. Budeme pokračovat z druhé zmíněné možnosti – tedy text jako komponenta.

Do vlastnosti **Text** napíšeme text a můžeme u toho změnit i pár dalších vlastností jako je písmo, zarovnání horizontální i vertikální, barvy a podobně. Změníme třeba velikost písma a vertikální zarovnání:



Když program spustíme, vidíme v levém dolním rohu text:



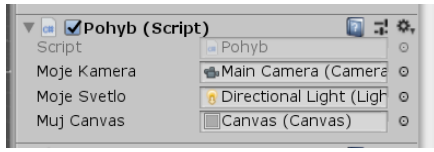
Počet sebraných mincí

Když skript **Mince** ohlásí skriptu **Pohyb**, že byla sebraná mince, chceme změnit zobrazený počet sebraných mincí. K tomu potřebujeme proměnnou, která bude obsahovat aktuální počet sebraných mincí a potřebujeme měnit text UI komponenty **Text**.

Abychom se dostali k vlastnosti **Text** objektu **Canvas**, přidáme skriptu proměnnou

```
public Canvas mujCanvas;
```

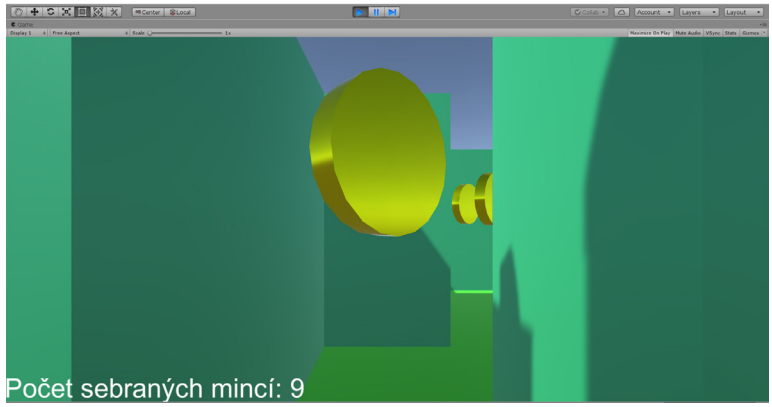
a v editoru vybereme Hrdinu a do proměnné skriptu **Pohyb** přetáhneme objekt **Canvas**:



Ve funkci **MinceSebrana()** potom najdeme komponentu **Text** a zapíšeme do ní počet sebraných mincí:

```
public void MinceSebrana()
{
    pocetSebranychMinci++;
    UnityEngine.UI.Text txt = mujCanvas.GetComponent<UnityEngine.UI.Text>();
    txt.text = $"Počet sebraných mincí: {pocetSebranychMinci}";
}
```

Když teď chodíme bludištěm a sbíráme mince, program zobrazuje, kolik jsme jich už sebrali:

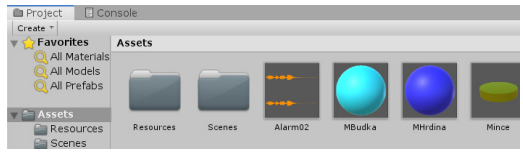


Další možnosti

Možností, jak vytvořit uživatelské rozhraní, je víc. Možností, jak udělat ten či onen krok, je také víc – berte to jako vzorek a experimentujte.

5.9 Zvuky

Zvuky, stejně jako obrázky, animace, materiály, skripty a další patří mezi **Assets**. Protože si zvuky chceme jenom vyzkoušet a protože pracujeme pod Windows, sáhneme do adresáře **C:\Windows\Media** (jestli máte Windows instalované na jiném disku nebo v jiném adresáři, použijte odpovídající jinou cestu a pokud používáte jiný operační systém, vyberte si jiný adresář se zvukovými soubory), popadneme například soubor **Alarm02.wav** a přetáhneme ho do okna **Assets**:



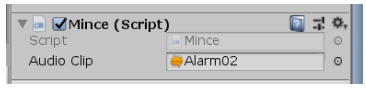
Soubor se zkonvertuje do objektu typu **AudioClip** a kdybychom chtěli, můžeme mu v okně Inspektoru nastavit další vlastnosti. Nechceme.

Abychom mohli zvuk přehrát při sebrání mince, přidáme do skriptu **Mince** veřejnou proměnnou

```
public AudioClip audioClip;
```

Potom v okně Assets vybereme prefab **Mince** (pozor, prefab! Ne stejnojmenný skript!):

a v okně Inspektoru v komponentě **Mince (Script)** kliknutím na ikonku nebo přetažením z okna Assets vyplníme proměnnou **AudioClip**:



Pak už stačí jen do funkce **Seber(GameObject mince)** přidat přehrání zvuku s určením místa, odkud bude zvuk vycházet:

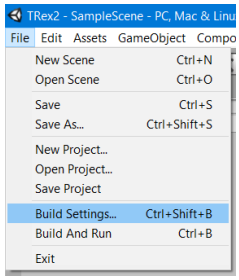
```
public void Seber(GameObject mince)
{
    sbiranaMince = mince;
```

```
        casDoKonceSbirani = 1f;
        AudioSource.PlayClipAtPoint(audioClip, transform.position);
    }
```

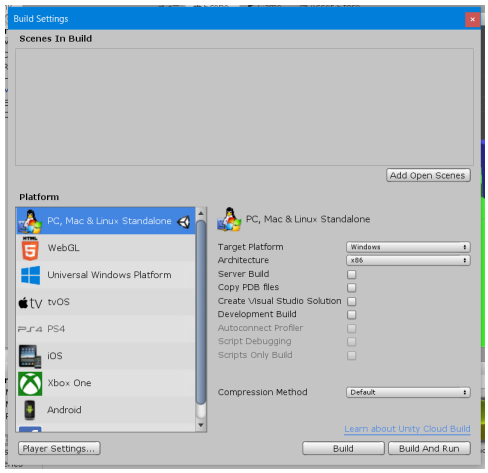
5.10 Sestavení

Naši hru zatím zkoušíme tak, že klikneme na tlačítko **Play** nebo použijeme klávesovou zkratku **Ctrl-P**. Ale kdybychom ji chtěli dát někomu, kdo nemá Unity, potřebujeme ji sestavit (překlad anglického **Build**).

V menu vybereme **File** a **Build Settings**:

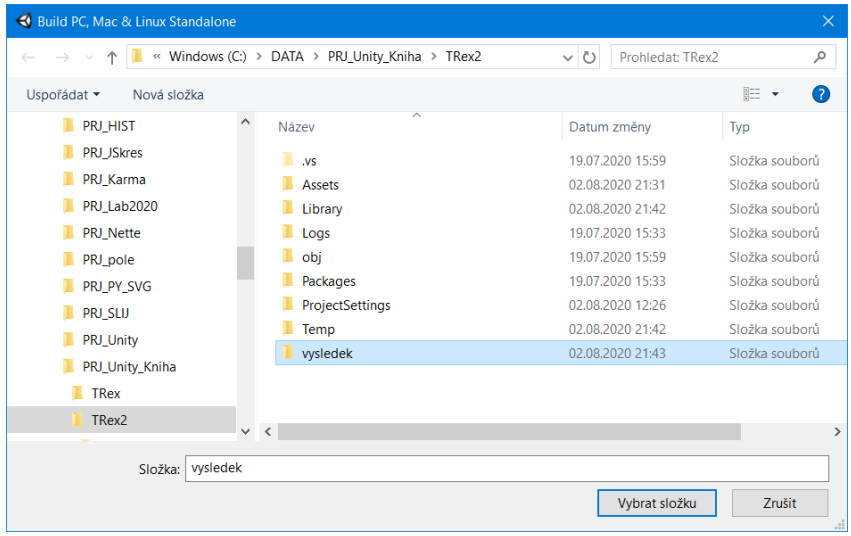


a vybereme si, na jaké platformě naše hra poběží (pro některé platformy může být potřeba doinstalovat nějakou část Unity):



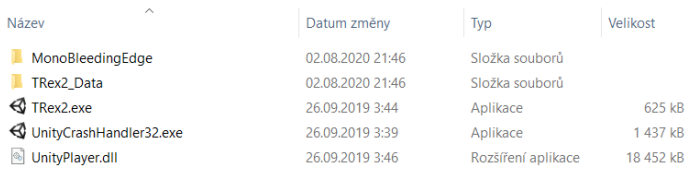
Protože chceme verzi pro Windows, nic neměníme a zvolíme **Build** nebo **Build And Run**.

Následuje dialog pro výběr adresáře, vytvoříme nový a pojmenujeme ho **vysledek**:



Klikneme na **Vybrat složku** a můžeme sledovat, jak probíhá překlad, může to trvat několik sekund.

Po skončení výsledný adresář obsahuje několik podadresářů a několik souborů – a spustitelný soubor s názvem stejným jako název našeho projektu je naše hra:



5.11 Kde je tyranosaurus?

No, není… nebo slovy jednoho starého vtipu – „Přece v Polsku!“ Tahle hra nám pomohla ukázat některé nástroje a tyranosaura si necháme na další hru.

6 Projekt Tyranosaurus

6 Projekt Tyranosaurus

6.1 Návrh

Pojďme vymyslet novou hru, a místo lepení po kouskách zkusme nejdříve dopředu rozmyslet, jak bude vypadat:

- bude na veliké ploše
- budou tam (náhodně rozmístěné) překážky
- bude mít více scén – tedy že se neskočí hned do hry, ale bude předtím nějaká úvodní scéna, pak vlastní hra a na konci nějaké výsledky
- bude tam hrdina
- kamera se bude dívat z pohledu první osoby, ale bude možné se pomocí myši dívat do stran, případně nahoru a dolů
- bude tam tyranosaurus, možná i víc tyranosaurů
- tyranosaurus nebude úplně hloupý a když uvidí hrdinu, půjde přímo k němu (ale když ho neuvidí, nechá ho být)
- tyranosaurus se bude pohybovat rychleji než hrdina, aby se před ním nedalo utéct
- když se tyranosaurus dotkne hrdiny, způsobí mu poškození nebo ubere jeden z jeho životů
- když hrdinovi nebude zbývat žádný život, hra končí
- hrdina bude moci tyranosaura na chvíli zastavit, pokud bude mít patřičné kouzlo/efekt/zbraň
- kouzla/efekty/zbraně budou na ploše náhodně rozmístěné, možná budou i přibývat a hrdina je bude nacházet a sbírat
- kromě kouzel/efektů/zbraní bude hrdina sbírat i mince, aby měl nějakou motivaci se pohybovat
- pohyb hrdiny bude fungovat pomocí přidávání síly a fyziky, takže žádná políčka a mřížka

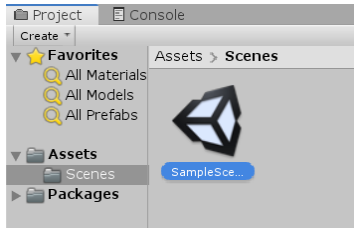
Máme návrh, uvidíme, jak to dopadne.

6.2 Nový projekt

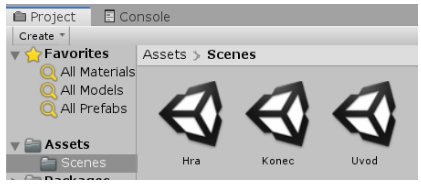
Spustíme Unity Hub, klikneme na **NEW**, případně vybereme verzi Unity (teď na tom nezáleží, vyhoví jakákoliv verze), vyplníme název projektu a cestu, kde bude uložen a potvrdíme **CREATE**.

6.3 Scény

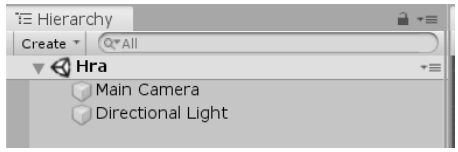
V okně **Assets** vidíme oddělení **Scenes**, když se do něj proklikáme, vidíme že obsahuje jedinou scénu **SampleScene**:



Smažeme ji a vytvoříme tři nové scény, které pojmenujeme **Uvod**, **Hra** a **Konec**:

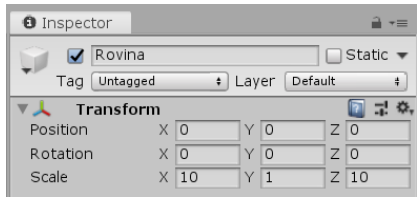


Dvojitým kliknutím vybereme scénu **Hra** a ostatní scény necháme na později. Která scéna je aktivní, můžeme vidět v okně Hierarchie:

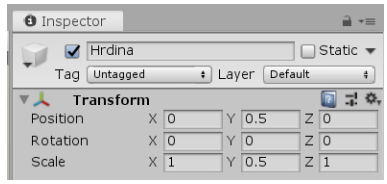


6.4 Objekty ve scéně

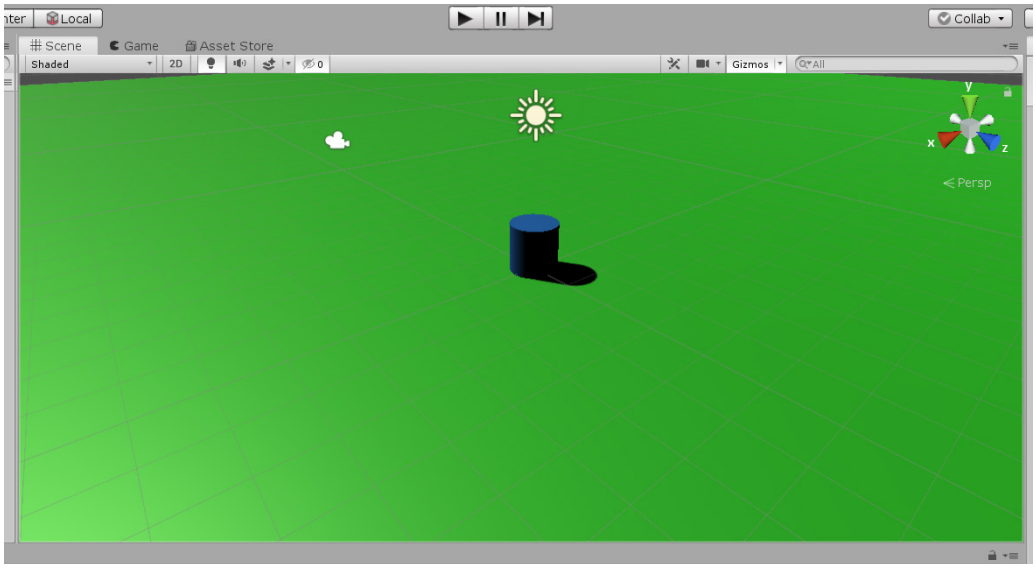
Pro to, po čem budou naši hrdinové chodit, můžeme použít buďto **Plane** nebo **Terrain**. Terrain dovede více, jako třeba editovat výšku, kopce a údolí, pěstovat stromy a podobně a zřejmě je o něco náročnější na vykreslování i detekci kolizí; pro naši hru vystačíme s rovinou (**Plane**) pojmenovanou **Rovina**. A protože chceme naši hru větší, tak jí desetkrát zvětšíme v ose X a v ose Z:



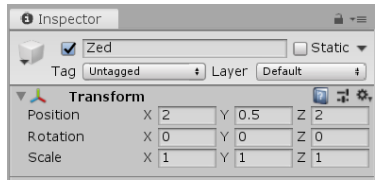
Hrdina (pojmenujeme ho tak) bude reprezentován zase válcem (Cylinder), kterému upravíme výšku, aby nebyl tak vysoký a polohu, aby se nevznášel nad zemí:



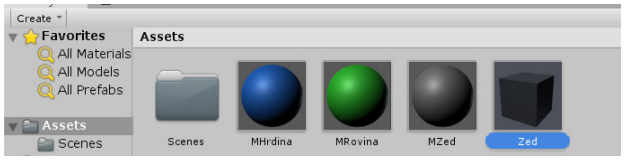
Stejně jako v minulé hře vyrobíme materiály **MRovina** a **MHrdina**, kterým zatím nastavíme jenom barvu:



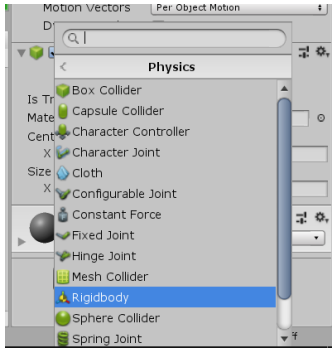
A podobně jako v minulé hře vyrobíme prefab **Zed**, ze kterého budeme vyrábět překážky a k němu materiál **MZed**,



pokud si už nepamatujete, jak se vyrobí prefab, tak tím, že nějaký objekt z okna Hierarchie přetáhneme mezi Assets, ale nejdříve se v okně Assets přepneme ze seznamu scén zase do kořenového adresáře (také bychom si mohli vyrobit složku na prefaby, ale vzhledem k tomu, kolik jich budeme mít (málo), bych to neřešil):



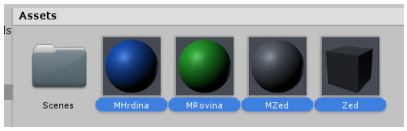
Představoval jsem si, že na začátku Zdi tak nějak napadají z výšky a tím vytvoří překážky, za kterými se budeme skrývat před tyranosaurem, ale když zkusíme tu instanci prefabu **Zed**, kterou máme ve scéně, zvednout a spustit hru, nic se neděje. Nepadá, protože nemá komponentu **Rigidbody** a tedy na ni nepůsobí gravitace. Tak ji přidáme –



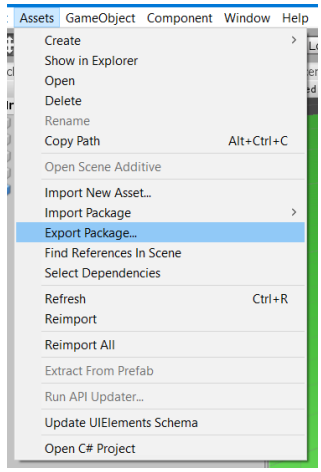
– a už padá.

6.5 Assets - export a import

Časem budeme chtít přidat ještě objekt **Mince**, včetně materiálu **MMince** a to už bychom se měli ptát, jestli je opravdu potřeba v novém projektu stejné assets vytvářet znovu. Potřeba to není, v okně Assets můžeme vybrat prvky, které chceme

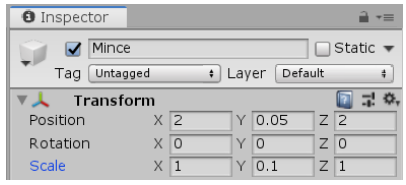


a v menu **Assets** potom říci, že je chceme vyexportovat:

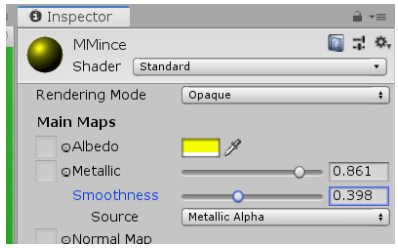


a v cílovém projektu zase ze stejného menu vybereme položku **Import Package**. Ale tento projekt chci vytvářet od začátku, bez závislostí, tak nic přebírat nebudeme.

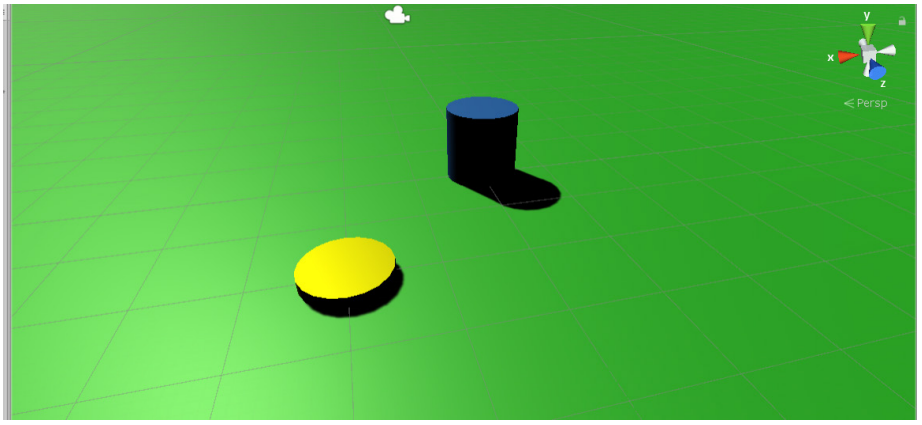
Minci vytvoříme podobně jako v minulém projektu – **3D Object – Cylinder**, přejmenujeme na **Mince**, upravíme rozměry



a vyrobíme žlutý lesklý materiál **MMince**

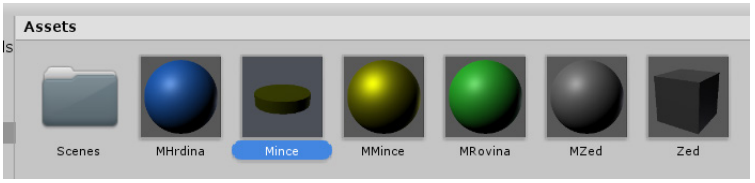


a přetáhneme ho na Minci:



(Tady jsme si pootočili pohled na scénu, abychom se dívali proti světlu a bylo vidět, jak se Mince leskne.)

Nakonec objekt **Mince** přetáhneme z okna **Hierarchie** do okna **Assets** a tím z něj vyrobíme prefab:

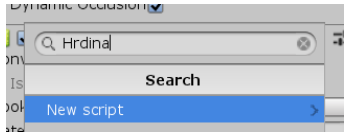


6.6 Skripty

Potřebujeme jeden skript, který by pohyboval Hrdinou a také pohyboval kamerou, když se hýbe Hrdina, a nakonec otáčel kamerou, když budeme hýbat myší. A druhý skript, který na začátku zaneřádí Rovinu instancemi prefabu Zed. Tak pojďme do toho!

Hrdina

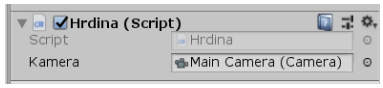
Skript pro pohyb Hrdiny tentokrát pojmenujeme **Hrdina**, stejně jako objekt, ke kterému bude patřit. Vyrobíme ho třeba tak, že vybereme Hrdinu ve scéně a v okně Inspektoru mu přidáme komponentu, nevybíráme z nabídky, ale napíšeme jméno a potvrdíme **New script** a pak ještě jednou:



Hrdina bude hýbat kamerou, tak do třídy přidáme veřejnou proměnnou

```
public Camera kamera;
```

a v okně Inspektoru do ní, po výběru objektu **Hrdina**, přetáhneme kameru **Main Camera**:

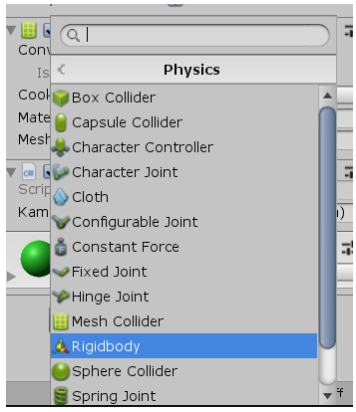


Pohyb Hrdiny i kamery budeme řešit ve funkci **Update()**, tentokrát bychom chtěli, aby:

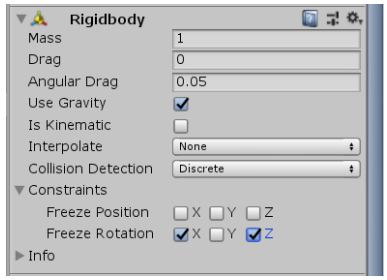
- šipka nahoru přidávala rychlost ve směru pohledu
- (možná i šipka dolů ubírala rychlost ve směru pohledu?)
- pohyb myší doleva nebo doprava měnil směr pohledu – a tím i směr působení síly
- pohyb myší nahoru a dolů měnil úhel pohledu nahoru a dolů

Přidávání a ubírání rychlosti budeme řešit působením síly metodou **AddForce()** a na konci nastavíme pozici a směr kamery podle pozice a směru Hrdiny tak, jak jsme to dělali v minulém projektu.

Ale abychom mohli na **Hrdinu** působit silou, potřebuje komponentu **Rigidbody**, proto mu ji přidáme:



a aby se nám při nárazu Hrdina nepřeklopil (spolu s ním by se překlopila i kamera), zafixujeme mu rotaci podle os X a Z, takže se bude moci otáčet pouze podle svislé osy **Y**:



Ve skriptu potom potřebujeme znát komponentu **Rigidbody**, abychom na ni mohli působit silou – a abychom ji nemuseli hledat pokaždé, budeme si ji pamatovat v proměnné, kterou naplníme v metodě **Start()**:

```
Rigidbody rb;

// Start is called before the first frame update
void Start()
{
    rb = GetComponent<Rigidbody>();
}
```

Ovládání

Pokud jde o ovládání pomocí kláves, to už známe – použijeme funkci **Input.GetKey()** s parametrem **„up“** (případně **„down“**). S myší to bude složitější a nemůže za to Unity.

Funkce **Input.GetAxis()** s parametrem **„Mouse X“** resp. **„Mouse Y“** vrátí rozdíl, o kolik se pohnula myš v příslušném směru od posledního zavolání. A při příštím zavolání už bude vracet nulu, dokud nepohneme myší.

My údaje o pohybu myši potřebujeme pro dva různé účely.

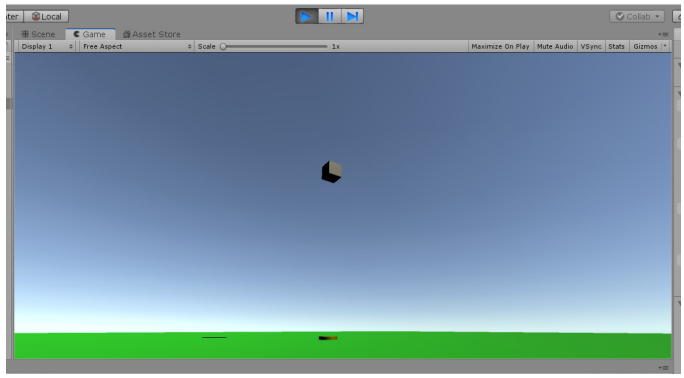
Když hráč pohne myší doleva nebo doprava, tak patřičně pootočíme Hrdinu, a tím jsme ten pohyb spotřebovali a je dobře, že příště bude nulový.

Když hráč pohne myší nahoru nebo dolů, chceme změnit úhel kamery, přesněji zarotovat ji podle osy **X** a nechat ji tak otočenou, dokud hráč zase nepohne myší. Protože se kamera bude otáčet

i do stran podle otáčení hráče, uděláme to tak, že si v proměnné mimo funkci **Update()** budeme pamatovat celkové pootočení směrem vzhůru a pohyb myši k němu budeme jenom přičítat:

```
float rotaceOsyX = 0f;

// Update is called once per frame
void Update()
{
    float SILA = 1000f;
    float POHLED = 5f;

    if (Input.GetKey("up"))
    {
        rb.AddForce(SILA * Time.deltaTime * transform.forward);
    }
    float rotaceOsyY = Input.GetAxis("Mouse X");
    rotaceOsyX += Input.GetAxis("Mouse Y");
       // X a Y jsou spravne - doleva/doprava rotuje kolem osy Y
    transform.Rotate(0, POHLED * rotaceOsyY, 0); // otocit se doleva/doprava


    kamera.transform.position = transform.position;
    kamera.transform.rotation = transform.rotation;
    kamera.transform.Rotate(POHLED * -rotaceOsyX, 0, 0);
}
```

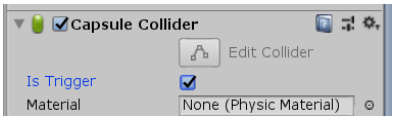
Takže se dokážeme Hrdinou přemisťovat po **Rovině** a myší měnit směr pohledu i směr, kterým budeme tlačeni silou (zkuste si třeba dojet na kraj… a nespadnout).



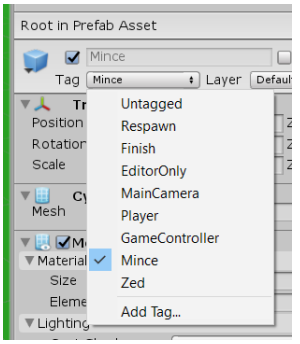
6.7 Sbírání mincí a detekce kolizí

Viděli jsme, že když objektu přidáme **collider**, tak na něj bude působit fyzika a dokáže poznat, že se srazil s jiným objektem, a patřičně reagovat. Alespoň zatím…

Collidery (nemám český překlad a nechci ho násilně vytvářet) mohou být různých druhů; ty, které zatím jsou v naší hře, jsou **CapsuleCollider** u Hrdiny a u Mince, **BoxCollider** u Zdi a **MeshCollider** u Roviny. Když při pohybu Hrdinou narazíme do Mince, zarazí se o ni (no, možná se začne pohybovat, o tom později). Ale v některých případech nepotřebujeme, aby collider bránil pohybu a stačí nám jenom, když se dozvíme, že došlo ke kolizi. Jako právě u Mince, kde můžeme Minci sebrat, vydat zvuk a nechat Hrdinu pokračovat v cestě. Dosáhneme toho tím, že u dotyčného colliderů zaškrtneme vlastnost **IsTrigger**, jako teď u prefabu **Mince** (u prefabu, protože chceme, aby to platilo pro všechny Mince):



A protože tušíme, že se budeme potřebovat zeptat, s kým se kdo srazil, rozšíříme si množinu značek (**tag**) a přiřadíme prefabům Mince a Zed odpovídající tagy:



a objektu Hrdina již existující tag **Player**.

Když nějaký objekt, jako třeba instance prefabu **Mince**, má nějaký collider, bude se mu volat funkce **OnCollisionEnter()**, případně funkce **OnTriggerEnter()**, tedy pokud bude mít připojený nějaký skript a pokud tam tu funkci napíšeme.

Takže když prefabu **Mince** přidáme komponentu skript **Mince** a do něj přidáme následující funkci:

```
private void OnTriggerEnter(Collider other)
{
    GameObject kdoKoliduje = other.gameObject;
    Debug.Log($"Mince.Triger: tag: {kdoKoliduje.tag}");

    gameObject.SetActive(false);
}
```

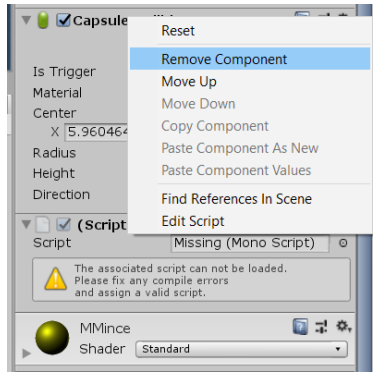
skript vypíše při nárazu Hrdiny do Mince zprávu

```
Mince.Triger: tag: Player
UnityEngine.Debug:Log(Object)
Mince:OnTriggerEnter(Collider) (at Assets/Mince.cs:16)
```

...a dotyčná Mince zmizí.

Při kolizi se volá odpovídající funkce u obou kolidujících objektů a protože sebrané mince bude počítat Hrdina, skript **Mince** můžeme zase smazat…

...tedy pokud to uděláme, tak Unity pozná, že se prefab Mince odkazuje na neexistující skript a vyzve nás, abychom ten odkaz odstranili – což uděláme v okně Inspektoru:



...a místo toho přidáme funkci do skriptu Hrdina:

```
private void OnTriggerEnter(Collider other)
{
    GameObject kdoKoliduje = other.gameObject;
    Debug.Log($"Hrdina.Triger: tag: {kdoKoliduje.tag}");
```

```
        kdoKoliduje.SetActive(false);
    }
```

která vypíše analogickou zprávu a zničí objekt, který dostala jako parametr.

```
Hrdina.Triger: tag: Mince
UnityEngine.Debug:Log(Object)
Hrdina:OnTriggerEnter(Collider) (at Assets/Hrdina.cs:47)
```

A protože by takto mohl Hrdina ničit všechny objekty, Zdi ani Tyranosaura nevyjímaje, přidáme k ničení ještě podmínku a také počítání sebraných mincí:

```
private void OnTriggerEnter(Collider other)
{
    GameObject kdoKoliduje = other.gameObject;
    Debug.Log($"Hrdina.Triger: tag: {kdoKoliduje.tag}");

    if (kdoKoliduje.tag == "mince")
    {
        pocetSebranychMinci++;
        kdoKoliduje.SetActive(false);
    }
}
```

Poznámka: *Funkce* ***SetActive()*** *ve skutečnosti objekt nezničí, jen zneaktivní; další možnosti uvidíme časem.*

Po zvýšení počtu sebraných mincí by měl následovat test, jestli už nejsou sebrané všechny, a případně ohňostroje, fanfáry a konec hry – ale na to je ještě brzy.

6.8 Příprava Roviny

Na začátku hry potřebujeme připravit Rovinu, podobně jako v minulé hře, tedy

- rozmístit Zdi
- rozmístit Mince
- umístit Hrdinu
- umístit Tyranosaura, až nějakého budeme mít, případně více Tyranosaurů

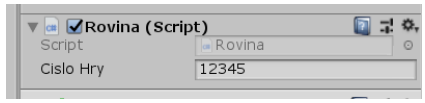
Použijeme na to skript **Rovina**, který připojíme k objektu **Rovina** – a jeho funkci **Start()**.

Náhoda

Všechny zmíněné objekty budeme umisťovat náhodně, aby hra byla pokaždé jiná, na druhou stranu by nebylo špatné dovolit hráči, aby si znovu zahrál stejnou hru, případně hru, kterou hrál jeho kamarád. Vyřešíme to tak, že náhodnému generátoru budeme na začátku nastavovat **náhodné semínko** – a to si bude (později) hráč moci zvolit. Se stejným semínkem bude náhodný generátor generovat stejnou posloupnost náhodných čísel, ze kterých postavíme stejnou scénu.

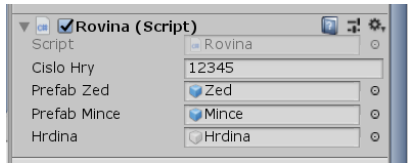
Takže objektu **Rovina** přidáme komponentu skript, který pojmenujeme **Rovina**, a odpovídajícímu objektu přidáme celočíselnou proměnnou:

```
public int CisloHry = 12345;
```



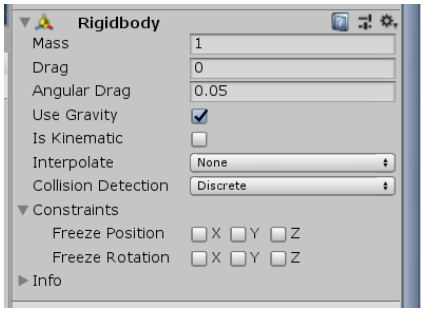
Protože budeme ve skriptu potřebovat ještě znát prefaby **Zed** a **Mince**, vytvoříme proměnné i pro ně a v okně Inspektoru do nich přetáhneme odpovídající prefaby. A nakonec totéž i pro **Hrdinu**, kterého tam také přetáhneme:

```
public int CisloHry = 12345;
public GameObject PrefabZed;
public GameObject PrefabMince;
public GameObject Hrdina;
```



Generování objektů

Zdi (i Mince i Hrdinu) rozmístíme náhodně a abychom neumístili dvě Zdi do stejného místa, a abychom to nemuseli dopředu hlídat, budeme je pouštět z výšky a spoléhat na to, že se při pádu nějak srovnají. Aby ovšem padaly, musíme prefabu **Zed** přidat komponentu **Rigidbody**; zatím neměníme žádné její vlastnosti, možná časem:



Hodilo by se nám také udělat ohrádku na okraji Roviny, aby Hrdina nemohl přepadnout. Ale Rovina je tentokrát větší a když bychom skládali ohrádku z jednotlivých kostek zdí, bude jich hodně a budou zbytečně zatěžovat procesor i paměť. Proto pro každou stranu ohrádky použijeme jenom jednu instanci prefabu **Zed**, kterou patřičně roztáhneme.

A když už budeme měnit vlastnosti instance, zvětšíme i její váhu (vlastnost **mass**), aby se nestalo, že padající Zdi ohrádku odsunou:

```
void Start()
{
    Collider collider = GetComponent<Collider>();
    Vector3 velikost = collider.bounds.size;

    // spoleham na to, ze velikost.x==velikost.z
    float minxz = -velikost.x / 2+1;
    float maxxz =  velikost.x / 2-1;
    float kamy = 0.5f;

    GameObject zed;
    zed = Instantiate(PrefabZed, new Vector3(0, kamy, minxz), Quaternion.identity);
    zed.transform.localScale = new Vector3(velikost.x, 1, 1);
    zed.GetComponent<Rigidbody>().mass = 1000;

    zed = Instantiate(PrefabZed, new Vector3(0, kamy, maxxz), Quaternion.identity);
    zed.transform.localScale = new Vector3(velikost.x, 1, 1);
    zed.GetComponent<Rigidbody>().mass = 1000;

    zed = Instantiate(PrefabZed, new Vector3(minxz, kamy, 0), Quaternion.identity);
    zed.transform.localScale = new Vector3(1, 1, velikost.z - 4);
    zed.GetComponent<Rigidbody>().mass = 1000;
```

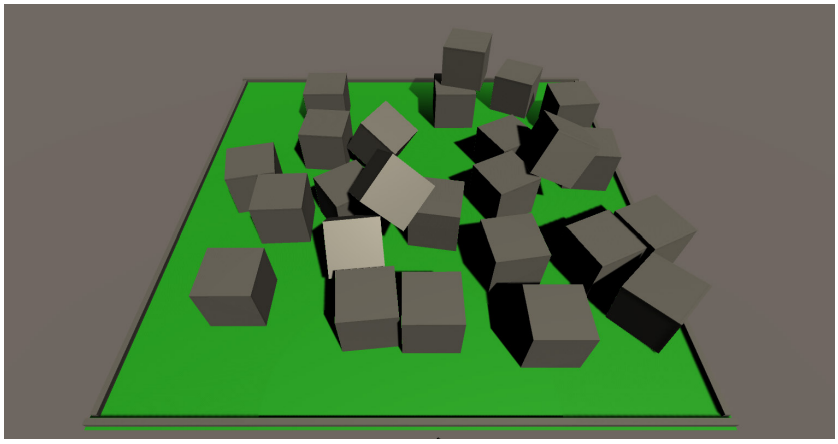
```
zed = Instantiate(PrefabZed, new Vector3(maxxz, kamy, 0), Quaternion.identity);
zed.transform.localScale = new Vector3(1, 1, velikost.z - 4);
zed.GetComponent<Rigidbody>().mass = 1000;
```

Na Rovinu chceme také přidat instance prefabu **Zed** jako překážky, které budou Hrdinovi (i Tyranosaurovi) bránit ve výhledu – proto nám nezáleží přesně na tom, jak budou rozmístěny a budeme je rozmisťovat pomocí náhodného generátoru, který nastavíme podle **Čísla hry**.

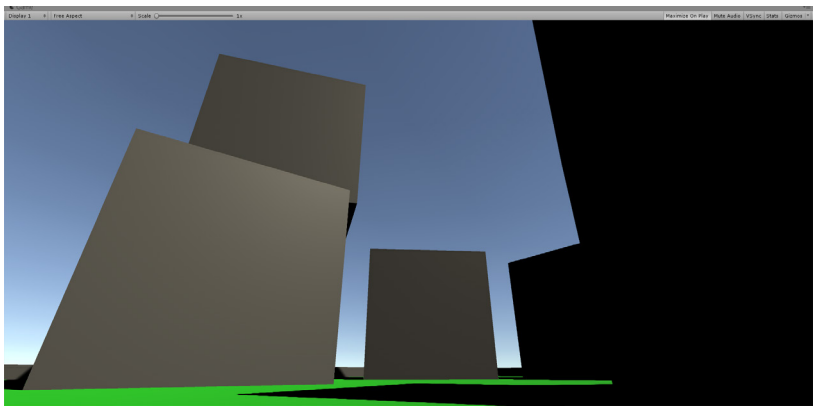
Abychom ale nevytvářeli příliš mnoho instancí, tak podobně jako u zdí tvořících ohrádku budeme měnit velikost vytvořené instance, řekněme na desetinásobek:

```
Random.InitState(CisloHry);
float K = 10f;

int pocetPrekazek = (int)(0.3 * velikost.x / K * velikost.z / K);
for (int i = 0; i < pocetPrekazek; i++)
{
    float x = Random.Range(minxz+K, maxxz-K);
    float z = Random.Range(minxz + K, maxxz - K);
    float y = (i + 2) * K; // nechame je padat z vysky

    zed = Instantiate(PrefabZed, new Vector3(x, y, z), Quaternion.identity);
    zed.transform.localScale = new Vector3(K, K, K);
}
```

Výsledek při pohledu z výšky vypadá takhle:



a při pohledu hrdiny takhle:



Je to dost strašidelné?

6.9 Poloha Mincí a Hrdiny

Bude potřeba někam umístit Mince a Hrdinu a nemělo by to být v místě, kde je nějaká **Zed**.

Kdybychom jednotlivé Zdi umisťovali na pevné pozice, tak bychom si mohli pamatovat jejich polohy a místa pro Mince a pro Hrdinu potom hledat tak, aby nekolidovala s žádnou zdí. Jenomže když spoléháme na fyziku a necháme jednotlivé Zdi padat a kutálet se, tak nedokážeme ve chvíli generování říci, kde která Zed nakonec bude stát. Co s tím?

Je to hra, ne řízení jaderné elektrárny, tak si můžeme dovolit zkusit řešení, které třeba někdy selže. Jestli chceme, aby se Mince a Hrdina dostali k Rovině až potom, co tam budou všechny Zdi, necháme je také padat z výšky a to z větší výšky, než všechny Zdi.

Co se může stát nejhoršího?

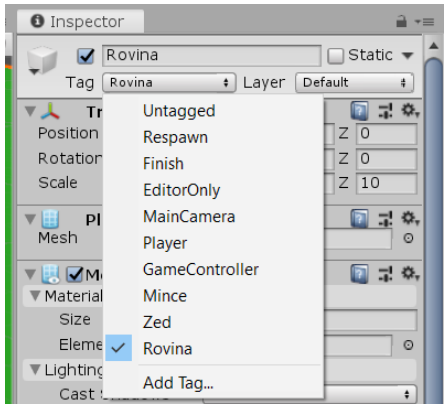
Pokud Hrdina dopadne na nějakou Zed, tak může skočit dolů, tedy až se vynadívá; berme to jako nečekaný bonus.

Pokud Mince dopadne na nějakou Zed, mohla by být pro Hrdinu nedostupná a jestli bude mít Hrdina za úkol najít všechny Mince, hra by nešla vyhrát. Můžeme to ale vyřešit tak, že počet Mincí, které Hrdina musí sebrat, bude na začátku nastavený na nulu a bude se zvyšovat o jedna pokaždé, když se nějaká Mince dotkne země (Roviny).

Pokud by Hrdina nebo Mince dopadli na dvorek obklopený Zdmi, ze kterého není úniku, pak máme smůlu, ale neřídíme jadernou elektrárnu...

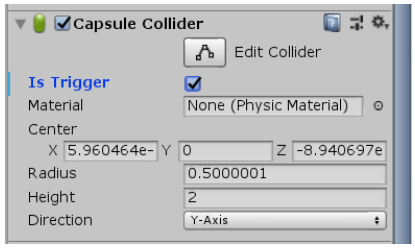
Vytvoření Mincí a umístění Hrdiny

Když vytvoříme Mince vysoko nad instancemi Zdí – tak tam zůstanou! Protože nemají komponentu **Rigidbody** a tedy nepadají. Takže ji přidáme, opět bez jakýchkoliv změn parametrů. A když už jsme v okně Inspektoru, tak vytvoříme nový tag **Rovina** a nastavíme ho objektu **Rovina**, aby se Mince mohla zeptat, kam to dopadla:

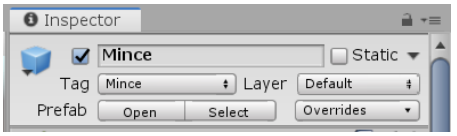


Problém: Propadávání

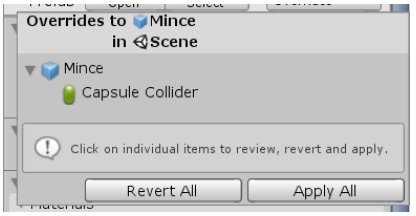
Když se podíváme na padající Mince, můžeme pozorovat, že (některé) propadnou **Rovinou** a padají dál. První důvod je zaškrtnutá vlastnost **IsTrigger**,



která říká, že tento collider nemá kolize sám řešit a má nám jenom dát vědět. Takže tuto vlastnost (zaškrtli jsme ji na samém začátku) zase odškrtneme a jestli jsme tu změnu udělali jen u jedné instance a ne u prefabu, můžeme v okně Inspektoru kliknout na tlačítko **Overrides**,

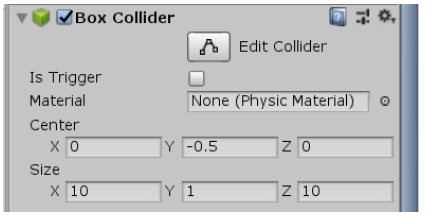


které ukáže, čím se tato instance liší od prefabu:



a můžeme si vybrat buďto všechny změny vrátit (**Revert All**) nebo naopak všechny změny promítnout do prefabu (**Apply All**) – a to je to, co chceme.

Pokud nám pořád některé mince dál propadávají Rovinou, pomůže odebrat Rovině **MashCollider** a přidat jí **BoxCollider**, kterému navíc změníme výšku a potom o polovinu výšky (MashCollider měl výšku nulovou) posuneme umístění dolů, aby nezastavoval objekty půl své výšky nad povrchem Roviny:



Skript Mince

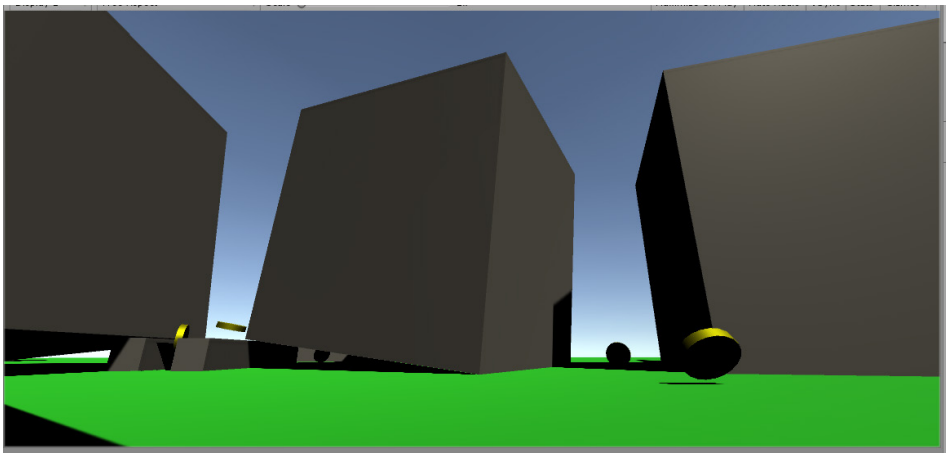
Minci potom přidáme skript **Mince** a protože její collider už není trigger, bude kolize vyvolávat jinou funkci.

Teď už by mince propadat neměly, ale pro všechny případy ponecháme Minci i funkci **Update()** a pokud by nějaká Mince propadla Rovinou nebo přepadla přes okraj, v určité hloubce ji zlikvidujeme a zmenšíme celkový počet mincí k sebrání. Ušetříme tím strojový čas:

```
public class Mince : MonoBehaviour
{
    static int ZbyvaSebrat = 0;

    void OnCollisionEnter(Collision collision)
    {
        GameObject kdoKoliduje = collision.gameObject;

        if (kdoKoliduje.tag == "Rovina")
        {
            ZbyvaSebrat++;
        }
        if (kdoKoliduje.tag == "Player")
        {
            ZbyvaSebrat--;
            //TODO: zahrat zvuk
            Destroy(gameObject);
        }
    }

    // Update is called once per frame
    void Update()
    {
        if (transform.position.y < -10)
        {
            ZbyvaSebrat--;
            Destroy(gameObject);
        }
    }
}
```

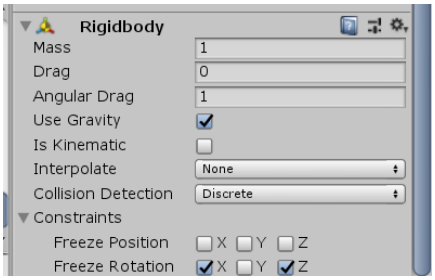
Poznámka: *Proměnná* ***ZbyvaSebrat*** *je označena jako* ***static****, protože to není proměnná, kterou by si pamatovala každá Mince, ale jedna společná proměnná pro celou třídu.*

Ještě odstraníme jednu instanci Zdi, která je od začátku umístěná ve scéně – a můžeme hru spustit:



6.10 Problém: Otáčení při nárazu

Pokud bychom se setkali s tím, že se při nárazu do jiného objektu Hrdina začne otáčet, pomůže mu v komponentě **Rigidbody** zvýšit úhlové tření (**Angular Drag**) z původních **0.05** třeba na **1**:



Takže máme překážky, mince, hrdinu, můžeme hrát. Vypadá to docela strašidelně – a to si ještě představte, že se tu někde potuluje tyranosaurus!

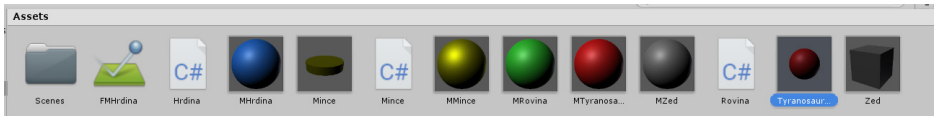
7 Objekt Tyranosaurus

7 Objekt Tyranosaurus

7.1 Příprava

Tyranosaura nejdříve vyrobíme, pak ošetříme, aby Hrdina reagoval na kolizi s ním – a až to všechno bude fungovat, naučíme Tyranosaura chodit.

Kdybychom chtěli odbočovat, mohli bychom si Tyranosaura vymodelovat v Blenderu, nejspíše bychom nějakého také našli ke stáhnutí. Protože nechceme odbočovat, bude Tyranosaurus jenom koule přejmenovaná na **Tyranosaurus**, trochu zvětšená a s červeným materiálem **MTyranosaurus**. A protože možná bude Tyranosaurů více, přetáhneme ho z okna Hierarchie do okna Assets a tím z něj vyrobíme prefab:

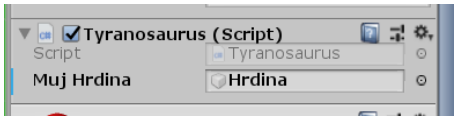


Teď uděláme několik (programovacích) kroků, které využijeme k zapojení Tyranosaura do hry, ale které budeme muset ošetřit jinak, až bude Tyranosarů více. Včas na to upozorníme.

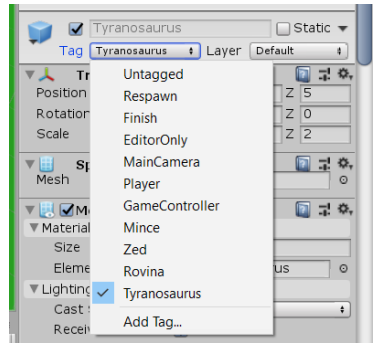
Aby se Tyranosaurus mohl pohybovat, přidáme mu komponentu skript pojmenovanou také **Tyranosaurus** a protože bude potřebovat vědět, kde je Hrdina, přidáme do skriptu parametr

```
public class TYranosaurus : MonoBehaviour
{
    public GameObject mujHrdina;
```

a v okně Inspektoru do něj přetáhneme Hrdinu (tady bude později problém, dojdeme k tomu):



A nakonec ještě rozšíříme množinu značek a prefabu(!) Tyranosaurus přidáme tag **Tyranosaurus**:



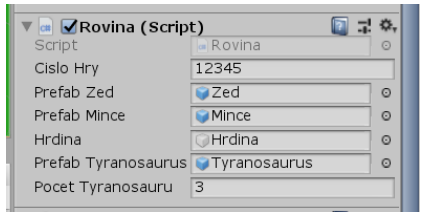
7.2 Zapojení do hry

Začneme tím, že se objekt Tyranosaurus nebude pohybovat, jenom ho/je ve skriptu **Rovina** vytvoříme a umístíme do scény a také naučíme Hrdinu reagovat na kolizi s Tyranosaurem.

Když už budeme ve skriptu Rovina vyrábět a umisťovat Tyranosaura, můžeme jich rovnou vyrobit a umístit více, takže skriptu Rovina přidáme veřejnou proměnnou **PocetTyranosauru** s předvyplněnou hodnotou a také proměnnou **PrefabTyranosaurus**, abychom mohli vyrábět nové instance:

```
public class Rovina : MonoBehaviour
{
    public int CisloHry = 12345;
    public GameObject PrefabZed;
    public GameObject PrefabMince;
    public GameObject Hrdina;
    public int PocetTyranosauru = 3;
    public GameObject PrefabTyranosaurus;
```

a v okně Inspektoru do ní přetáhneme odpovídající prefab:



Vlastní vytvoření a umístění potom bude podobné jako vytvoření a umístění Hrdiny, souřadnice **X** a **Z** budeme generovat náhodně a hotové Tyranosaury budeme shazovat z výšky a doufat, že se s pomocí fyziky nějak srovnají.

Mohli bychom ale ohlídat, abychom nevytvořili Tyranosaura příliš blízko Hrdiny, aby Hrdina nebyl sežrán dříve než se stačí rozhlédnout, proto využijeme toho, že si při vytváření pamatujeme souřadnice Hrdiny, a u vygenerovaných souřadnic pro Tyranosaura budeme kontrolovat, jestli nejsou moc blízko (i když se tam stejně můžou po pádu dokutálet):

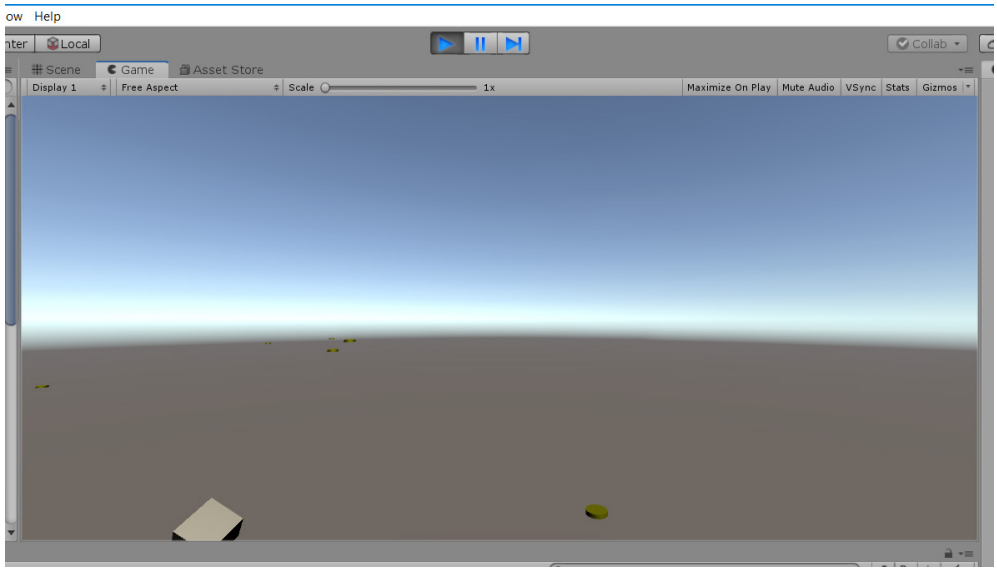
```
        float minimalniVzdalenostOdHrdiny = 10f;
        Vector2 hrdinaXZ = new Vector2(hx, hz);
        for (int i = 0; i < PocetTyranosauru; i++)
        {
```

```
        float tx;
        float tz;

        do
        {
            tx = Random.Range(minxz + 1, maxxz - 1);
            tz = Random.Range(minxz + 1, maxxz - 1);
        } while (Vector2.Distance(new Vector2(tx, tz), hrdinaXZ)
                    > minimalniVzdalenostOdHrdiny);

        float ty = hy + 5 + 2 * i;
        Instantiate(PrefabTyranosaurus, new Vector3(tx, ty, tz), Quaternion.identity);
    }
```

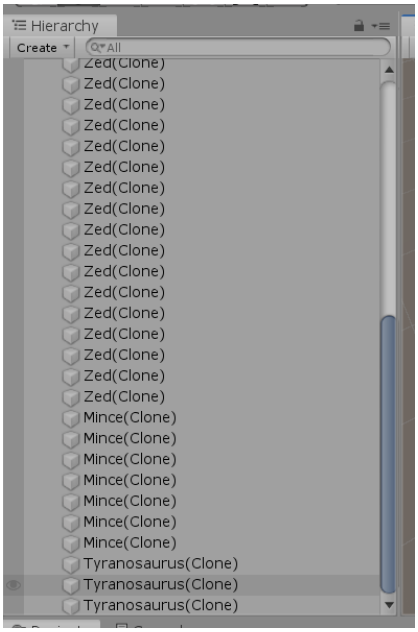
7.3 Ladění v Unity

Když hru spustíme, můžeme vidět, jak padají Zdi, padá Hrdina, padají Mince – a nepadají žádní Tyranosauři! Jistě, je to proto, že nemají komponentu **Rigidbody** a tedy na ně nepůsobí gravitace, ale ukážeme si u toho, jak nám Unity může pomoci při ladění.

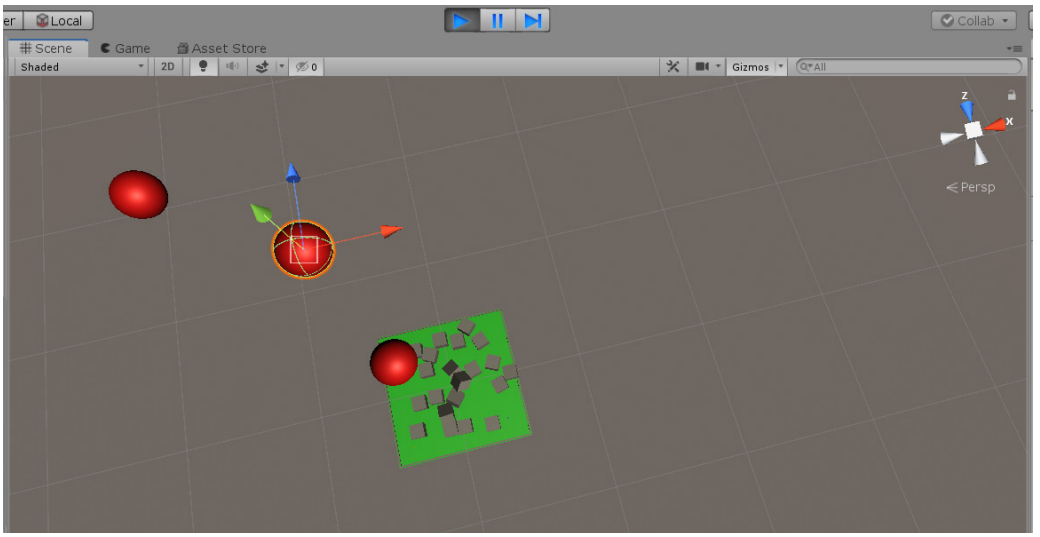
Když hru spustíme, možná máme a možná nemáme zamáčknutý přepínač **Maximize On Play**:



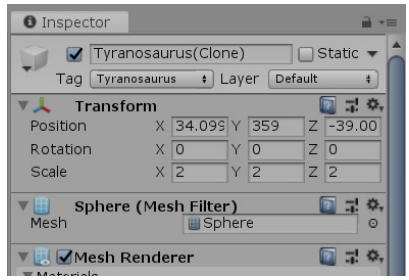
Pokud máme, tak si ho od-máčkneme a to nám dovolí v běžící hře používat stejné nástroje jako při vytváření scény, třeba okno Hierarchie, kde vidíme všechny objekty:



(u opravdového programu bychom si měli připravit prázdné objekty – přihrádky na různé typy objektů a vyrábět instance do nich, ne takhle všechny vedle sebe!). Vidíme i okno Scény, kde můžeme hýbat kamerou i jednotlivými objekty:

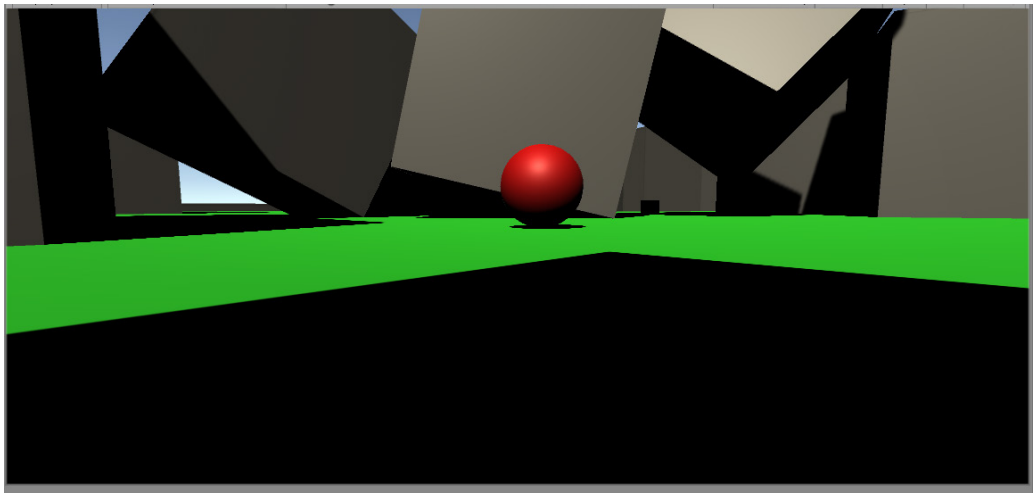


A nakonec i okno Inspektoru, kde můžeme vidět i měnit vlastnosti vybraného objektu:



7.4 Zpátky k Tyranosaurům

Takže prefabu Tyranosaurus přidáme komponentu **Rigidbody** a teď už ve hře můžeme narazit na Tyranosaura:

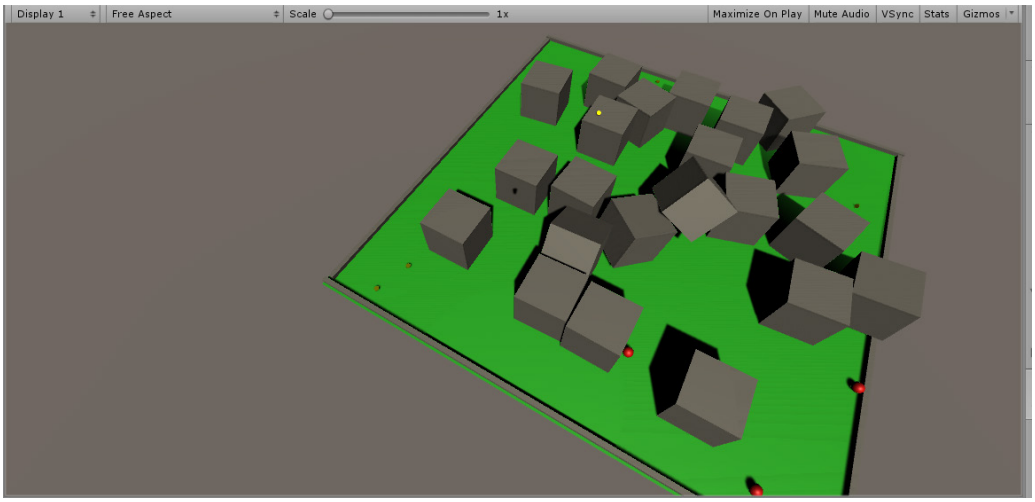


Když na něj Hrdina opravdu narazí, tak se Tyranosaurus odkutálí, proto upravíme skript Hrdiny:

```
void OnCollisionEnter(Collision collision)
{
    GameObject kdoKoliduje = collision.gameObject;
    if (kdoKoliduje.tag == "Tyranosaurus")
```

```
        {
            //TODO: zvuk
            rb.velocity = (50*Vector3.up);
        }
    }
```

Tyranosaurus nakonec hráči nebude ubírat život, jenom ho vystřelí do výšky, čímž hráč ztratí nějaký čas, na druhou stranu se bude moci z výšky rozhlédnout a případně i sebrat nějaké mince, které nedopadly na zem:



7.5 Pohyb Tyranosaura

Tyranosaurus nemusí být zvlášť inteligentní. V každém kroku se podívá, jestli vidí Hrdinu – a pokud ano, vydá se k němu. Pokud ne, tak bude pokračovat k místu, kde naposledy viděl Hrdinu. Pokud už tam došel nebo Hrdinu ještě neviděl, vybere si náhodně směr a vydá se tam, jako by tam předtím viděl Hrdinu.

Problém: Parametr skriptu u prefabu

Teď přijde ta změna, na kterou jsme upozorňovali výše při vytváření skriptu Tyranosaurus:

Skript **Tyranosaurus** potřebuje přístup k objektu **Hrdina**. Když jsme ho před chvílí nastavovali pro konkrétní instanci Tyranosauta, tak nebyl problém, jen jsme vybrali a přetáhli objekt Hrdiny

nebo kliknuli na ikonku a vybrali z menu. Ale když to chceme nastavit u prefabu, tak to nejde. Proto odkaz Tyranosaura na Hrdinu nakonec budeme získávat až za běhu ve funkci **Start()**:

```
GameObject mujHrdina;
Vector3 kdeJsemNaposledyVidelHrdinu;
Rigidbody rb;

// Start is called before the first frame update
void Start()
{
    mujHrdina = GameObject.FindGameObjectWithTag("Player");

    kdeJsemNaposledyVidelHrdinu = Vector3.zero;
    rb = GetComponent<Rigidbody>();
}
```

Způsobů, jak najít objekt, je řada; tady se nám hodí možnost vyhledat objekt podle hodnoty vlastnosti **tag**. Konec problému.

Vlastní pohyb by se potom mohl řídit algoritmem popsaným výše, ale potřebujeme, aby se tak Tyranosaurus začal chovat teprve potom, co dopadne na Rovinu, proto přidáme proměnnou a reakci na kolize:

```
bool uzDopadl = false;
void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Rovina")
    {
        uzDopadl = true;
    }
}
```

Vlastní pohyb Tyranosaura potom bude vypadat takto:

```
// Update is called once per frame
void Update()
{
    if (uzDopadl == false)
    {
        return;
    }
```

```
        float RYCHLOST = 20f;
        float BLIZKO = 1f;
        float KROK = 2f;

        // jestli vidim Hrdinu:
        RaycastHit hit;
        Vector3 smer = mujHrdina.transform.position - transform.position;

        if ((Physics.Raycast(transform.position, smer, out hit))
            && (hit.collider.gameObject.tag == "Hrdina"))
        {
            kdeJsemNaposledyVidelHrdinu = mujHrdina.transform.position;
            rb.velocity = Time.deltaTime * RYCHLOST * smer.normalized;
        }
        else // pokracovat ve smeru, pokud tam jeste nejsem
        if ((transform.position-kdeJsemNaposledyVidelHrdinu).sqrMagnitude>BLIZKO)
        {
            smer = mujHrdina.transform.position - transform.position;
            rb.velocity = Time.deltaTime * RYCHLOST * smer.normalized;
        }
        else // uz jsem tam dosel a Hrdinu nevidim => zvolit si novy cil
        {
            float uhel = Random.Range(0f, Mathf.PI * 2);
            kdeJsemNaposledyVidelHrdinu
                = new Vector3(KROK * Mathf.Sin(uhel), 0, KROK * Mathf.Cos(uhel));
        }
    }
```

Proti původnímu plánu tyranosaurus není rychlejší než hrdina, ale i tak to stačí na strašidelnou atmosféru.

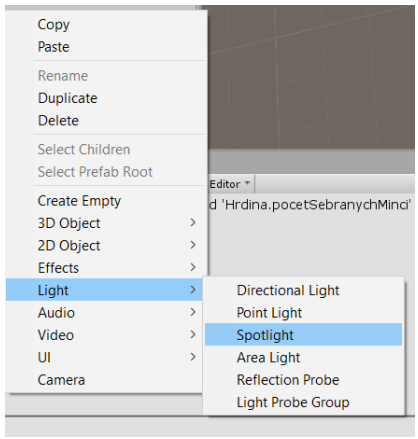
Poznámka: *Ta vlastnost* ***.normalized*** *vektoru* ***smer*** *vrací normalizovanou hodnotu – tedy vektor o velikosti jedna. Kdybychom ji nepoužili a nastavovali rychlost podle* ***smer****, znamenalo by to, že ke vzdálenějšímu Hrdinovi poběží Tyranosaurus rychleji a naopak k blízkému půjde po-ma-lu a skončí jako rychlonohý Achilles, který nemůže dohonit želvu.*

7.6 Světlo

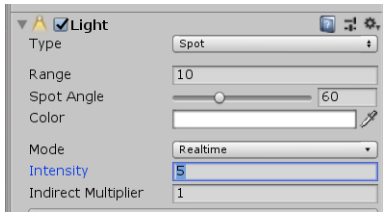
Když tak hrdina chodí po zákoutích naší hry a hledá mince – tak tam není moc dobře vidět.

Je to hra, tak si můžeme říci, že to je součást herního zážitku. Nebo mu můžeme dát do ruky baterku.

Světlo a práce s ním je obecně složitá záležitost, my uděláme jen to, že v okně Hierarchie vyrobíme nový objekt typu **Spotlight**:



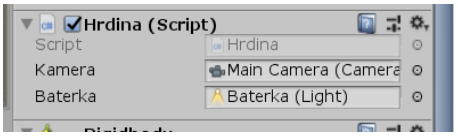
Pojmenujeme ho **Baterka**, trochu mu upravíme vlastnosti (zvětšíme úhel a intezitu světla):



do skriptu **Hrdina** pro něj přidáme veřejnou proměnnou

```
public Light baterka;
```

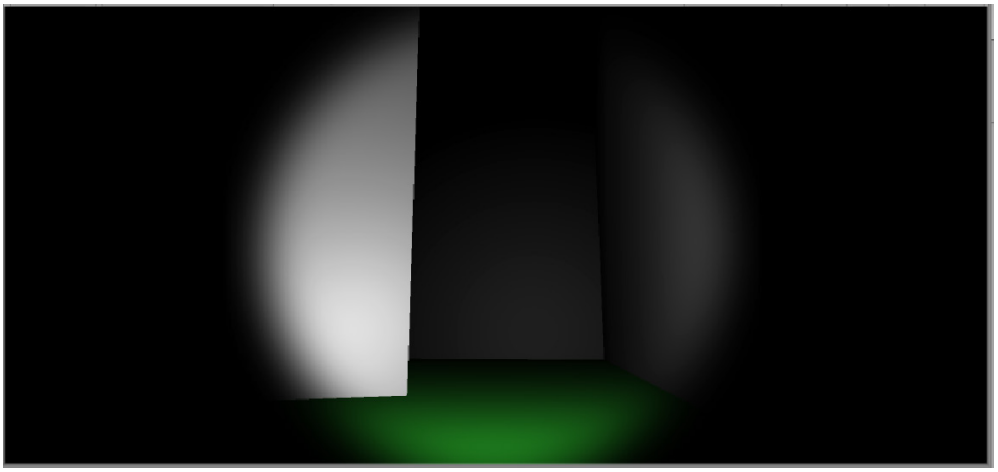
V okně Inspektoru do ní přetáhneme objekt **Baterka**:



a jako poslední krok – ve skriptu Hrdina tam, kde měníme pozici a orientaci kamery, budeme analogicky nastavovat i pozici a orientaci baterky, můžeme rovnou použít hodnoty z kamery:

```
kamera.transform.position = transform.position;
kamera.transform.rotation = transform.rotation;
kamera.transform.Rotate(POHLED * -rotaceOsyX, 0, 0);

baterka.transform.position = kamera.transform.position;
baterka.transform.rotation = kamera.transform.rotation;
```

Potom pohyb hrdiny v temných uličkách vypadá takhle:



ale možná je to změkčilost a také bychom to mohli zapínat a vypínat jenom když si to hrdina může dovolit, třeba na omezenou dobu nebo podle sebraných mincí.

Také můžeme experimentovat s nastavováním parametrů světla, nebo s jinými typy světel.

8 Scény

8 Scény

8.1 Zamyšlení

Na začátku práce na této hře jsme vyrobili tři scény a pak jsme se věnovali jen scéně **Hra**. Teď se pojďme podívat na ty ostatní a na přechody mezi nimi a možná si také trochu ujasnit pravidla a cíl hry.

Pravidla a cíl

Hrdina chodí v bludišti, hledá mince a utíká před tyranosaurem. Ale to není cíl, tak to zkusme zformulovat lépe. Sebrat všechny mince – se nám nemusí podařit, protože i když jsme si dali práci s počítáním, kolik mincí dopadlo na zem, některé mohou být nedostupné mezi (nebo pod) zdmi a naopak, díky „vykopnutí" od tyranosaura může hrdina sbírat i mince, které zůstaly ležet někde nahoře na zdi. Pokud tedy nemůžeme zaručit, že půjdou sebrat všechny mince, mohlo by být cílem sebrat jich co nejvíc. A aby se výsledky her daly porovnávat, zapojíme do toho ještě čas, tedy **sebrat co nejvíce mincí za co nejkratší dobu**. A když máme dvě kriteria, mince a dobu, není jasné, co je lepší; buďto bychom mohli výsledek brát jako dvojici (mince, doba) a porovnávat ho lexikograficky nebo můžeme stejného efektu dosáhnout tím, že to převedeme na jedno číslo, třeba takhle:

skore = 1000*sebrané_mince – doba

První scéna - Uvod

Na první scéně by si měl uživatel nastavit nebo vybrat parametry hry, tedy

- číslo hry (náhodné semínko)
- počet tyranosaurů
- případně jejich rychlost

Ale nastavovat si příliš mnoho parametrů není zábavné, navíc by bylo těžké mezi sebou porovnávat výsledky téže hry s jiným počtem nebo jinou rychlostí tyranosaurů. Proto to zase spojme do jednoho čísla, třeba tak, že se bude zadávat jenom číslo hry, přičemž první číslice bude udávat počet tyranosaurů a druhá jejich rychlost, takže třeba hra číslo **12345** bude obsahovat jediného tyranosaura s rychlostí dva.

Kromě pole (nebo seznamu) pro zadávání (nebo výběr) **čísla hry** už bude první hra obsahovat pouze **tlačítko**, které hru spustí.

A krom toho tam může být řada nefunkčních prvků jako návod k použití, obrázky apod.

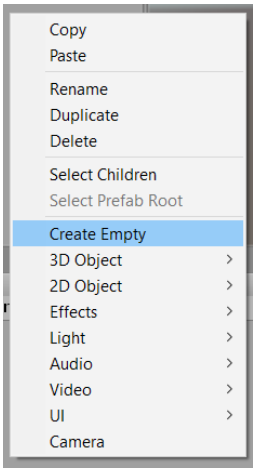
Poslední scéna - Konec

Poslední scéna by měla ukázat **dosažené skóre**, případně **žebříček nejvyšších skóre** pro hru daného čísla – a **tlačítko** pro přechod na scénu **Uvod**, aby uživatel mohl hrát znovu a případně si vybrat jinou hru.

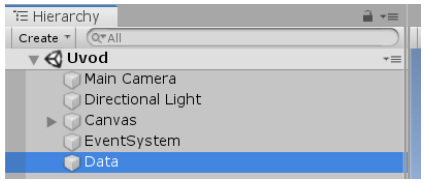
8.2 Přetrvávající objekty

V každé scéně se na začátku vytvoří její objekty – a při nahrání nové scény se zničí. Což je mrzuté, protože potřebujeme ze scény **Uvod** přenést vybrané číslo hry do scény **Hra** a z ní zase přenést výsledné skóre do scény **Konec**. Unity nabízí řešení v podobě funkce **DontDestroyOnLoad()**, kde parametrem je objekt, který se nemá ničit při přechodu mezi scénami.

V okně Hierarchie vyrobíme nový prázdný objekt:



a přejmenujeme ho na **Data**:



Objektu **Data** přidáme komponentu – skript **Data** a do něj přidáme veřejné proměnné pro hodnoty, které budeme chtít udržovat mezi scénami:

```
public class Data : MonoBehaviour
{
    public int CisloHry;
    public int SebranychMinci;
    public float DelkaHry;
```

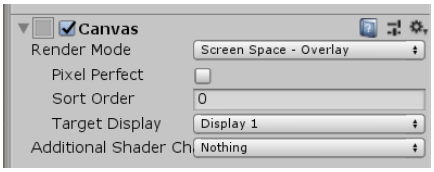
Aby objekt **Data** zůstal zachován mezi scénami, přidáme mu funkci **Awake()**, s voláním již zmíněné funkce **DontDestroyOnLoad**:

```
void Awake()
{
    DontDestroyOnLoad(this.gameObject);
}
```

Funkce **Awake()** se volá po načtení instance skriptu a zavolá se dříve než kterákoliv z funkcí **Start()**.

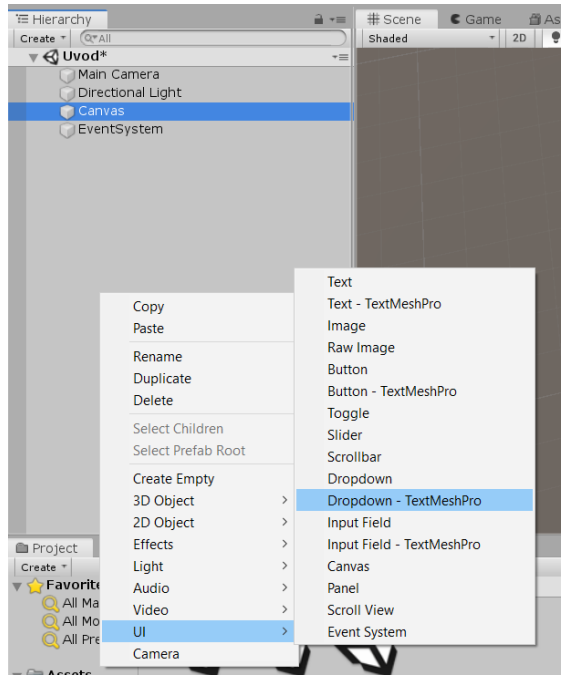
8.3 Zpátky k první scéně

Vybereme tedy první scénu **Uvod** a vložíme do ní ovládací prvky. Nejprve **Canvas**, kterému nastavíme režim vykreslování na **Screen Space – Overlay**:

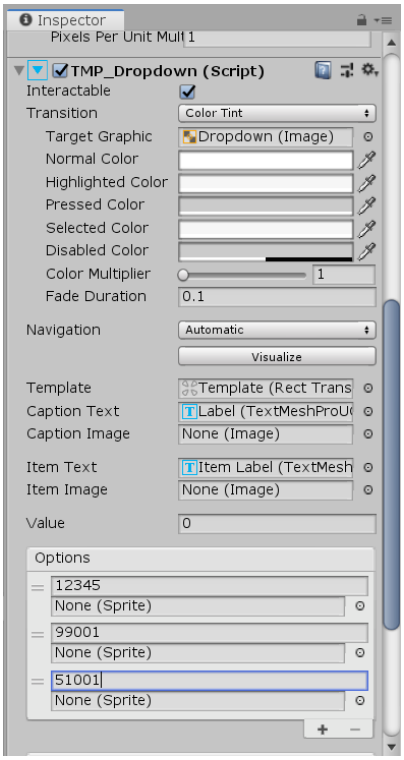


Číslo hry

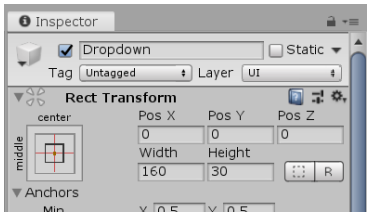
Dále v okně Hierarchie přidáme (můžeme ho přidat kamkoliv, ale zařadí se pod objekt Canvas) objekt **Dropdown – TextMeshPro** z kategorie **UI**:



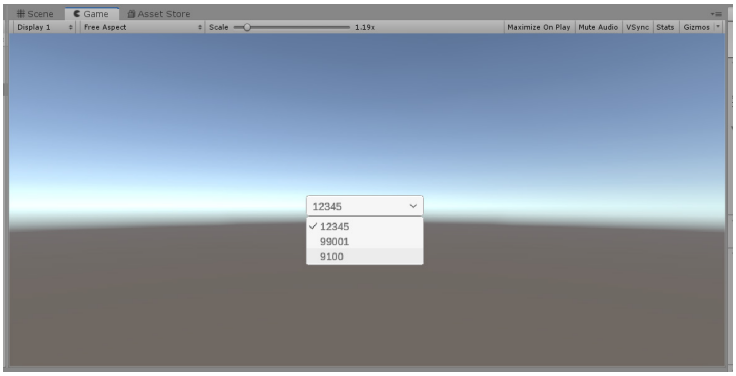
a v okně Inspektoru mu nastavíme několik možných čísel her jako hodnoty parametru **Options**:



Souřadnice nastavíme na střed obrazovky:



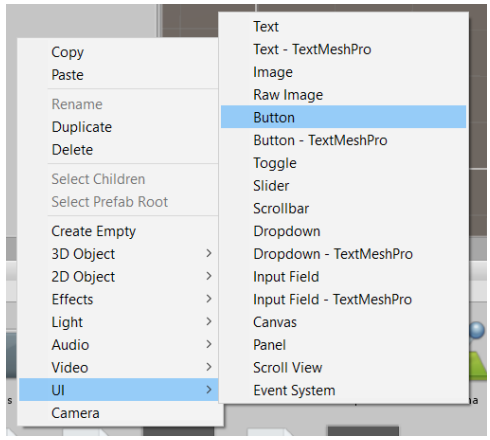
a když hru zkusíme spustit, máme vybírací seznam:



K tomu, jak získat vybrané číslo hry, se dostaneme později.

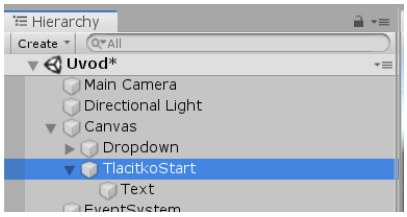
Tlačítko Start

Nakonec přidáme tlačítko (tentokrát ne z knihovny TexMesh Pro, i když to by šlo také):

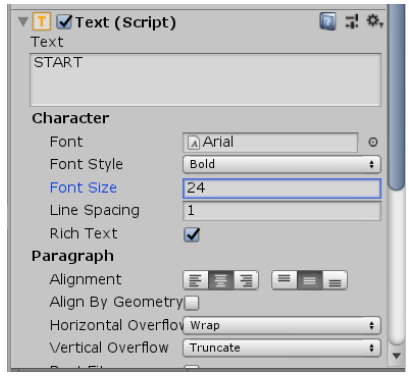


...a přejmenujeme ho na **TlacitkoStart.**

Pokud chceme změnit text, který je na něm napsán, všimneme si, že tlačítko v okně Hierarchie má podřízený objekt **Text**:



A ten objekt je to správné místo, kde změnit text zobrazený na tlačítku a případně i řadu jiných vlastností:



Pro tlačítko si vyrobíme skript, pojmenujeme ho **TlacitkoStart**.

Poznámka: *Pokud by nás napadlo pojmenovat si ten skript* ***Button****, jako to děláme u ostatních objektů, není to dobrý nápad,*

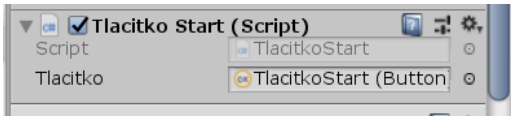
```
//!!! tohle NE !!!
public class Button : MonoBehaviour
{
    public Button tlacitko;
```

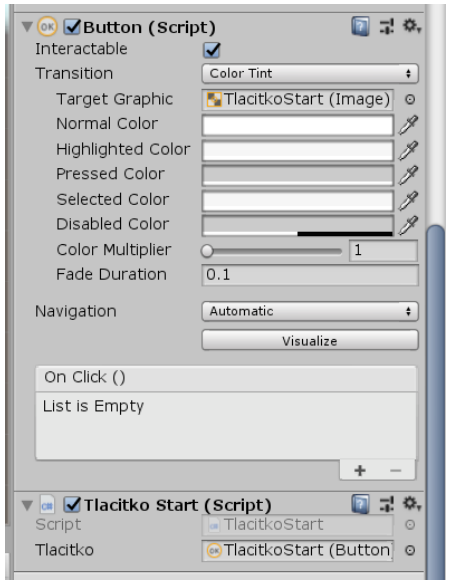
protože v takovém případě název třídy skriptu (Button) zastíní typ pro tlačítko (taky Button) a potom ten přidaný parametr ***tlacitko*** *nebude typu Button-tlačítko, ale Button-skript. Pomohlo by u proměnné* ***tlacitko*** *uvést typ celým jménem, tedy* ***UnityEngine.UI.Button*** *namísto* ***Button****, ale stejně je lepší si skript pojmenovat jinak.*

Novému skriptu přidáme veřejnou proměnou

```
public class TlacitkoStart : MonoBehaviour
{
    public UnityEngine.UI.Button tlacitko;
```

přetáhneme do ní objekt tlačítka:





Poznámka: *Samotný objekt typu* ***Button*** *už obsahoval komponentu-skript pojmenovaný* ***Button****, takže kdybychom ponechali původní název, může v tom navíc být trochu zmatek. České identifikátory někdy mají své výhody.*

Ve funkci **Start()** potom nastavíme, aby tlačítko při kliknutí zavolalo naši funkci:

```
void Start()
{
    tlacitko.onClick.AddListener(KLIK);
}
```

… a **KLIK()** je funkce, ze které nastartujeme hru.

K tomu budeme potřebovat ještě zjistit vybranou hodnotu v objektu **Dropdown…** ...jenomže nastartovat hru (a tedy přejít do jiné scény) je něco, co by se nemuselo odehrávat jen tak v nějaké funkci obsluhující kliknutí. Proto funkci pro start hry přidáme do skriptu **Data** a tady ji jenom zavoláme. Jenomže zase (vývoj programu není snadná záležitost!), aby se skript dostal k objektu **Data**, bude si ho muset najít:

```
void KLIK()
{
    Data data = FindObjectOfType<Data>();
    data.GetComponent<Data>().ZacniNovouHru();();
}
```

Šlo by to napsat i bez proměnné, do jednoho příkazu, ale takhle mi to přijde přehlednější.

Ta volaná funkce vypadá takhle:

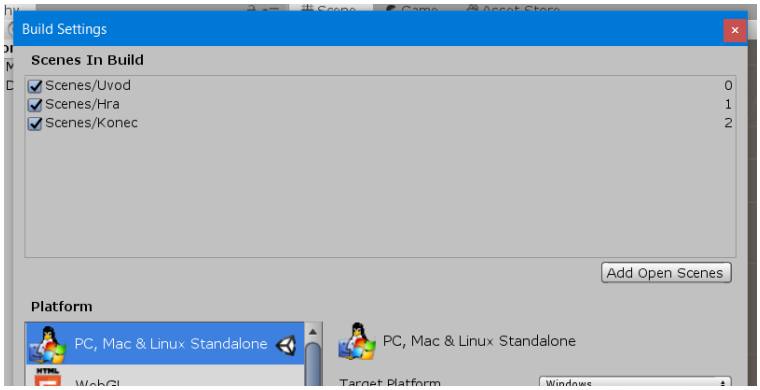
```
public void ZacniNovouHru()
{
    TMPro.TMP_Dropdown dropdown = FindObjectOfType<TMPro.TMP_Dropdown>();
    string sCisloHry = dropdown.options[dropdown.value].text;
    CisloHry = int.Parse(sCisloHry);
    PocetTyranosauru = int.Parse(sCisloHry.Substring(0, 1));    // prvni cislice
    RychlostTyranosaura = int.Parse(sCisloHry.Substring(1, 1)); // druha cislice

    UnityEngine.SceneManagement.SceneManager.LoadScene("Hra");
}
```

a nebude fungovat,

> Scene ‚Hra' couldn't be loaded because it has not been added to the build settings or the AssetBundle has not been loaded.
> To add a scene to the build settings use the menu File->Build Settings...

```
UnityEngine.SceneManagement.SceneManager:LoadScene(String)
Data:ZacniNovouHru() (at Assets/Data.cs:24)
TlacitkoStart:KLIK() (at Assets/TlacitkoStart.cs:19)
UnityEngine.EventSystems.EventSystem:Update()
```

protože jsme v okně **Build Settings** nenastavili, jaké scény se mají zahrnout do spouštěného programu:



Scény přidáváme po jedné, pomocí tlačítka **Add Open Scenes** (vždycky je potřeba patřičnou scénu otevřít), nemusíme to potvrzovat žádným z tlačítek, stačí okno jen zavřít křížkem.

A přidali jsme tam i scénu **Konec**, i když v ní zatím nic není.

8.4 Zapojení objektu Data

Rovina

Vytváření Mincí a Tyranosaurů ve skriptu **Rovina** by teď mělo brát v úvahu parametry hry uložené v objektu **Data**:

```
// Start is called before the first frame update
void Start()
{
    Data data = FindObjectOfType<Data>();
    CisloHry = data.CisloHry;
    PocetTyranosauru = data.PocetTyranosauru;
```

Rovina používá k nalezení dat funkci **FindObjectOfType<Data>()**.

Tyranosaurus

Podobně to udělá skript **Tyranosaurus**, jenom aby nemusel objekt **Data** hledat při každém volání funkce **Update()**, přesune si proměnnou **RYCHLOST** z funkce ven a nastaví její hodnotu ve funkci **Start()**:

```
float RYCHLOST = 20f;

// Start is called before the first frame update
void Start()
{
    Data data = GameObject.Find("Data").GetComponent<Data>();
    RYCHLOST = 10f * data.RychlostTyranosaura;
```

(Tyranosaurus využívá jinou možnost nalezení objektu dat.)

Mince

Když Hrdina sebere Minci – a to ve svém collideru ošetřuje Mince – měla by se připočítat k celkovému počtu sebraných mincí. Protože zároveň nakonec nepotřebujeme celkový počet mincí, které je nutno sebrat, nebude Mince řešit kolizi s Rovinou a funkce bude reagovat jen na Hrdinu:

```
void OnCollisionEnter(Collision collision)
{
    GameObject kdoKoliduje = collision.gameObject;
    if (kdoKoliduje.tag == "Player")
    {
        Data data = FindObjectOfType<Data>();
        data.SebranychMinci++;
        Destroy(gameObject);
    }
}
```

Kdybychom přidali zvuky, tak tady by se hodilo přehrát zvuk.

Hrdina

Hrdina by měl zvětšovat údaj o tom, jak dlouho hra trvá. Tedy, stačilo by uložit si, kdy hra začala, a na konci odečíst od aktuálního času, ale rozhodli jsme se to udělat takhle. Navíc bychom hráči/

Hrdinovi měli přidat možnost hru ukončit, třeba stiskem klávesy **Esc**. Obojí se bude odehrávat ve funkci **Update()**.

Protože k hodnotám objektu **Data** bude Hrdina přistupovat opakovaně, najde si ho jenom jednou v metodě **Start()** a zapamatuje si ho:

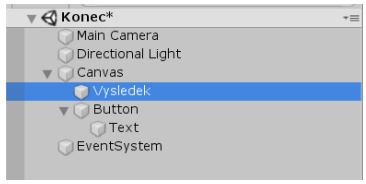
```
Data data;

// Start is called before the first frame update
void Start()
{
    data = FindObjectOfType<Data>();
    rb = GetComponent<Rigidbody>();
}
```

Ve funkci **Update()** potom bude přičítat čas a při stisku **Esc** ukončí hru:

```
// Update is called once per frame
void Update()
{
    data.DelkaHry += Time.deltaTime;

    if (Input.GetKeyDown(KeyCode.Escape))
    {
        UnityEngine.SceneManagement.SceneManager.LoadScene("Konec");
    }
```

8.5 Scéna Konec

Scéna Konec zatím neobsahuje nic, víme, že tam chceme zobrazit skóre a tlačítko pro přechod do scény **Uvod** (možná bychom chtěli i něco víc, ale teď ne), tak přidáme objekty **UI Text** a **UI Button**, objekt Canvas se u toho vytvoří automaticky. Objekt **Text** přejmenujeme na **Vysledek**, aby se nám lépe hledal v kódu:



Trochu si pohrajeme s písmem a velikostí, aby se objekty nepřekrývaly (jen trochu):

a tlačítku přidáme skript **TlacitkoZpet**, který jednak obslouží reakci na stisk a jednak do textu napíše výsledné skóre:

```
public class TlacitkoZpet : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        Data data = FindObjectOfType<Data>();
        int skore = 1000 * data.SebranychMinci - (int)data.DelkaHry;

        GameObject vysledek = GameObject.Find("Vysledek");
        vysledek.GetComponent<UnityEngine.UI.Text>().text = skore.ToString();

        GameObject tlacitko = GameObject.Find("Button");
        tlacitko.GetComponent<UnityEngine.UI.Button>().onClick.AddListener(KlikZpet);
    }

    void KlikZpet()
    {
        UnityEngine.SceneManagement.SceneManager.LoadScene("Uvod");
    }
}
```

Ty objekty **Button** a **Vysledek** bychom mohli předat jako parametr a nastavit v okně Inspektoru, ale jde to i takhle. Objekt **Data** si vyhledáme stejně, jako jsme to už dělali v jiných místech a přechod do jiné scény už jsme také viděli.

Když chvíli hrajeme a zmáčkneme klávesu **Escape**, vypadá to takhle:

8.6 Leccos tam schází

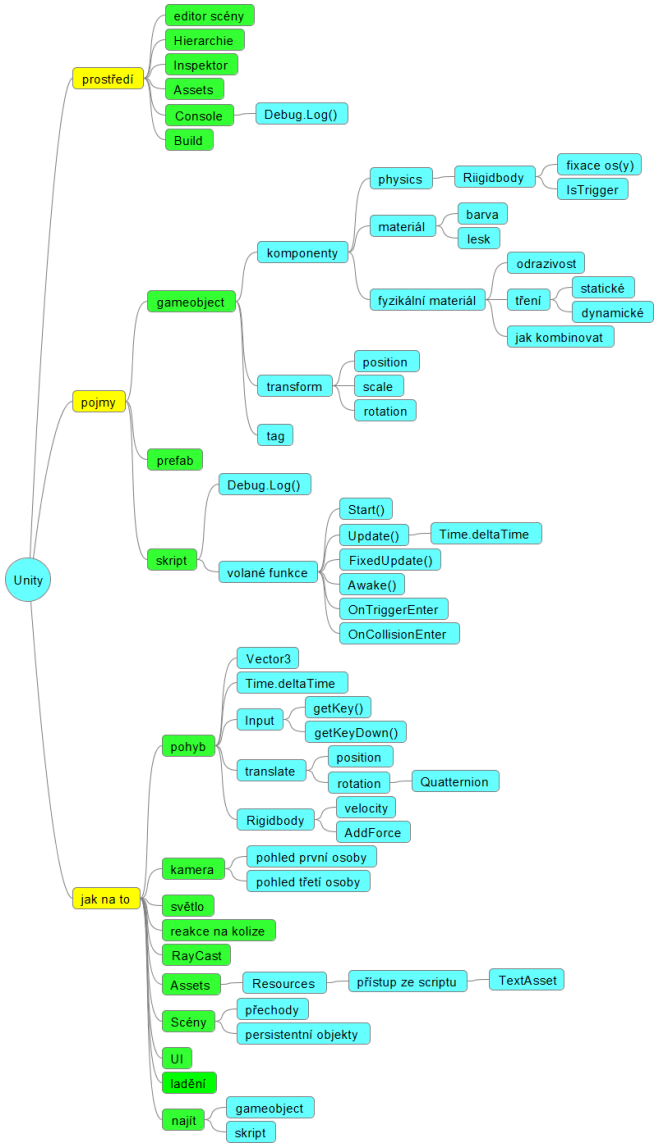
Schází tam přidat zvuky při sebrání mince, případně při útoku tyranosaura. Není tam ani ukládání a zobrazování nejlepších výsledků, na to by se hodil textový soubor nebo třeba SQL databáze. Chybí leccos ale všechno by už mělo být snadné nebo snadno dohledatelné; procházka končí, dál pokračujte sami!

P.S.: Všimli jste si, že tyranosarus dokáže postrkovat zdi? Jáma a kyvadlo!

Příloha 1: Mapa

Příloha 1: Mapa

Myšlenková mapa možná není nejlepší nástroj, protože struktura toho, co chci ukázat, není hierarchická – ale chtěl jsem vidět nějaký celkový pohled na to, o čem jsme mluvili



Příloha 2: Skripty

Příloha 2: Skripty

Kdyby někdo chtěl vidět pohromadě celé zdrojové texty, tak tady jsou, pro první (Hra) a druhý (Tyranosaurus) projekt.

1 Hra

Pohyb.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Pohyb : MonoBehaviour
{
    public Camera mojeKamera;
    public Light mojeSvetlo;
    public Canvas mujCanvas;

    // Start is called before the first frame update
    void Start()
    {
        inverseMoveTime = 1f / moveTime;
        rb = GetComponent<Rigidbody>();
    }

    float moveTime = .1f;
    Rigidbody rb;

    float inverseMoveTime;

    protected IEnumerator PlynulyPohyb(Vector3 konec)
    {
        float zbyvajiciVzdalenostsqr = (transform.position - konec).sqrMagnitude;

        while (zbyvajiciVzdalenostsqr > float.Epsilon)
        {
            Vector3 novaPozice
              = Vector3.MoveTowards(rb.position, konec, rychlostPohybu * Time.deltaTime);
            rb.MovePosition(novaPozice);
```

```
        zbyvajiciVzdalenostsqr = (transform.position - konec).sqrMagnitude;
        yield return null;
    }
}

bool probihaPohyb = false;
Vector3 cilovaPozice;
float zbyvajiciCas;
float rychlostPohybu = 5;

bool probihaOtaceni = false;
float smerOtaceni;
Vector3 cilovaOrientace;
float rychlostOtaceni = 5;

int pocetSebranychMinci = 0;

public void MinceSebrana()
{
    pocetSebranychMinci++;
    UnityEngine.UI.Text txt = mujCanvas.GetComponent<UnityEngine.UI.Text>();
    txt.text = $"Počet sebraných mincí: {pocetSebranychMinci}";
}

void Update()
{
    while (true)
    {
        if (probihaPohyb)
        {
            zbyvajiciCas -= Time.deltaTime;
            if (zbyvajiciCas <= 0)
            {
                transform.position = cilovaPozice;
                probihaPohyb = false;
            }
            else
            {
                transform.position
                    += Time.deltaTime * rychlostPohybu * transform.forward;
                break; // nebudeme se ptat na klavesy
```

```
            }
        }

        if (probihaOtaceni)
        {
            zbyvajiciCas -= Time.deltaTime;
            if (zbyvajiciCas <= 0)
            {
                transform.eulerAngles = cilovaOrientace;
                probihaOtaceni = false;
            }
            else
            {
                transform.Rotate(0, Time.deltaTime * rychlostOtaceni * smerOtaceni, 0);
                break; // nebudeme se ptat na klavesy
            }
        }

        if (Input.GetKey("up"))
        {
            RaycastHit hitInfo;

            //StartCoroutine(PlynulyPohyb(transform.position + transform.forward));
            //break;

            if (Physics.Raycast(transform.position, transform.forward, out hitInfo,
1f))
            {
                GameObject predeMnou = hitInfo.collider.gameObject;

                if (predeMnou.tag == "zed")
                {
                    Debug.Log("nejdu - prekazka");
                    break;
                }
                if (predeMnou.tag == "mince")
                {
                    Mince skriptMince
                       = GameObject.FindObjectOfType(typeof(Mince)) as Mince;
                    skriptMince.Seber(predeMnou);
                    break;
```

```
                    }
                }
                else
                    Debug.Log("RayCast nic nenasel");

                // kdyz jsem dosel sem, tak JDU a uz se na nic neptam:
                //StartCoroutine(PlynulyPohyb(transform.position + transform.forward));
                //break;


                if (Physics.Raycast(transform.position, transform.forward, out hitInfo, 1f)
                        == false)
                {
                    cilovaPozice = transform.position + transform.forward;
                    probihaPohyb = true;
                    zbyvajiciCas = 1f / rychlostPohybu;
                    //PlynulyPohyb(cilovaPozice);
                }
            }
            else
            if (Input.GetKey("left") || Input.GetKey("right"))
            {
                smerOtaceni = (Input.GetKey("left") ? -90 : 90);
                cilovaOrientace = transform.rotation.eulerAngles + smerOtaceni * Vector3.
up;
                probihaOtaceni = true;
                zbyvajiciCas = 1f / rychlostOtaceni;
            }
            break;
        }

        mojeKamera.transform.position = transform.position;
        mojeKamera.transform.rotation = transform.rotation;
    }
}
```

Rovina.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class Rovina : MonoBehaviour
{
    public GameObject PrefabZed;
    public GameObject PrefabMince;

    // Start is called before the first frame update
    void Start()
    {
        Collider collider = GetComponent<Collider>();

        // ziskat bludiste:
        string[] blud = BLUDISTE_M;

        // zjistit rozmery:
        int sx = blud[0].Length;
        int sz = blud.Length;

        // prizpusobit velikost Roviny - a posunout ji:
        transform.localScale = new Vector3(sx / 10.0f, 1, sz / 10.0f);
        transform.position += new Vector3((sx-1) / 2f, 0, (sz-1) / 2f);

        // nasazet Zdi a umistit a nasmerovat Hrdinu:
        GameObject hrdina = GameObject.Find("Hrdina");
        for (int x = 0; x < sx; x++)
        {
            for (int z = 0; z < sz; z++)
            {
                switch (blud[sz-1-z][x])
                {
                    case ZED:
                        Instantiate(PrefabZed, new Vector3(x, 1, z), Quaternion.identity)
                            .transform.localScale = new Vector3(1, 2, 1);
                        break;
                    case ,m‘:
                        Instantiate(PrefabMince, new Vector3(x, 1.2f, z),
                           Quaternion.Euler(90, 45, 0));
                        break;
                    case ,^‘:
                        hrdina.transform.position = new Vector3(x, 1, z);
                        hrdina.transform.eulerAngles = 0*Vector3.up;
                        break;
```

```
                case ,<‘:
                    hrdina.transform.position = new Vector3(x, 1, z);
                    hrdina.transform.eulerAngles = -90 * Vector3.up;
                    break;
                case ,>‘:
                    hrdina.transform.position = new Vector3(x, 1, z);
                    hrdina.transform.eulerAngles = 90 * Vector3.up;
                    break;
                case ,v‘:
                    hrdina.transform.position = new Vector3(x, 1, z);
                    hrdina.transform.eulerAngles = 180 * Vector3.up;
                    break;
            }
        }
    }
    return;
}

const char ZED = ,X‘;

string[] BLUDISTE_A =
{
        "XXXXXXXXXX",
        "X       <X",
        "X  XXXXXXX",
        "X  X     X",
        "X  XXX XXX",
        "X  X   X X",
        "X  X X   X",
        "X  XXXXX X",
        "X        X",
        "XXXXXXXXXX"
};

string[] BLUDISTE_M =
{
        "XXXXXXXXXX",
        "X m m m <X",
        "X  XXXXXXX",
        "X  X     X",
        "X  XXX XXX",
```

```
        "X  XmmmX X",
        "X  X XmmmX",
        "X  XXXXXmX",
        "X    mmmmX",
        "XXXXXXXXXX"
};

string[] Bludiste_A()
{
    return BLUDISTE_A;
}

string[] Bludiste_B()
{
    TextAsset ta = Resources.Load("BludisteB") as TextAsset;
    char[] odd = { ,\r‘, ,\n‘ };
    string[] blud = ta.text.Split(odd, System.StringSplitOptions.RemoveEmptyEntries);
    return blud;
}

string[] Bludiste_N()
{
    int sx = Random.Range(5, 10);
    int sz = Random.Range(5, 10);
    string[] blud = new string[sz];

    float hustota = 0.20f;
    bool uzBylHrdina = false;

    for (int z = 0; z < sz; z++)
    {
        blud[z] = "";
        for (int x = 0; x < sx; x++)
        {
            if ((x == 0) || (x == sx-1) || (z == 0) || (z == sz-1)
                    || (Random.value < hustota))
                blud[z] += ZED;
            else
            if (uzBylHrdina == false)
            {
                blud[z] += "^";
```

```
                    uzBylHrdina = true;
                }
                else
                    blud[z] += " ";
            }
        }
        return blud;
    }


    // Update is called once per frame
    void Update()
    {

    }
}
```

Mince.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Mince : MonoBehaviour
{
    float casDoKonceSbirani;
    GameObject sbiranaMince;
    public AudioClip audioClip;

    // Start is called before the first frame update
    void Start()
    {
        casDoKonceSbirani = 0f;
    }

    public void Seber(GameObject mince)
    {
        sbiranaMince = mince;
        casDoKonceSbirani = 1f;
        AudioSource.PlayClipAtPoint(audioClip, transform.position);
    }
```

```
    // Update is called once per frame
    void Update()
    {
        if (casDoKonceSbirani > 0)
        {
            sbiranaMince.transform.eulerAngles += Time.deltaTime * 720 * Vector3.up;

            //sbiranaMince.transform.position += Time.deltaTime * 3f * Vector3.up;
            casDoKonceSbirani -= Time.deltaTime;
            if (casDoKonceSbirani <= 0)
            {
                sbiranaMince.SetActive(false);

                Pohyb skriptPohyb = GameObject.FindObjectOfType(typeof(Pohyb)) as Pohyb;
                skriptPohyb.MinceSebrana();
            }
        }

    }
}
```

2 Tyranosaurus

Data.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Data : MonoBehaviour
{
    public int CisloHry;
    public int PocetTyranosauru;
    public float RychlostTyranosaura;
    public int SebranychMinci;
    public float DelkaHry;

    void Awake()
    {
        DontDestroyOnLoad(this.gameObject);
```

```
    }

    public void ZacniNovouHru()
    {
        TMPro.TMP_Dropdown dropdown = FindObjectOfType<TMPro.TMP_Dropdown>();
        string sCisloHry = dropdown.options[dropdown.value].text;
        CisloHry = int.Parse(sCisloHry);
        PocetTyranosauru = int.Parse(sCisloHry.Substring(0, 1));    // prvni cislice
        RychlostTyranosaura = int.Parse(sCisloHry.Substring(1, 1)); // druha cislice

        UnityEngine.SceneManagement.SceneManager.LoadScene("Hra");
    }
}
```

Hrdina.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Hrdina : MonoBehaviour
{
    public Camera kamera;
    public Light baterka;

    Rigidbody rb;
    Data data;

    // Start is called before the first frame update
    void Start()
    {
        data = FindObjectOfType<Data>();
        rb = GetComponent<Rigidbody>();
    }

    float rotaceOsyX = 0f;

    // Update is called once per frame
    void Update()
    {
        data.DelkaHry += Time.deltaTime;
```

```
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            UnityEngine.SceneManagement.SceneManager.LoadScene("Konec");
        }

        float SILA = 1000f;
        float POHLED = 5f;

        if (Input.GetKey("up"))
        {
            rb.AddForce(SILA * Time.deltaTime * transform.forward);
        }
        float rotaceOsyY = Input.GetAxis("Mouse X");
        rotaceOsyX += Input.GetAxis("Mouse Y");
            // X a Y jsou spravne - doleva/doprava rotuje kolem osy Y
        transform.Rotate(0, POHLED * rotaceOsyY, 0); // otocit se doleva/doprava

        kamera.transform.position = transform.position;
        kamera.transform.rotation = transform.rotation;
        kamera.transform.Rotate(POHLED * -rotaceOsyX, 0, 0);

        baterka.transform.position = kamera.transform.position;
        baterka.transform.rotation = kamera.transform.rotation;
    }

    int pocetSebranychMinci = 0;
    int pocetZivotu = 5;

    void OnCollisionEnter(Collision collision)
    {
        GameObject kdoKoliduje = collision.gameObject;

        if (kdoKoliduje.tag == "Tyranosaurus")
        {
            //TODO: zvuk
            rb.velocity = (50*Vector3.up);
        }
    }
}
```

Mince.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Mince : MonoBehaviour
{
    static int ZbyvaSebrat = 0;

    void OnCollisionEnter(Collision collision)
    {
        GameObject kdoKoliduje = collision.gameObject;

        if (kdoKoliduje.tag == "Rovina")
        {
            ZbyvaSebrat++;
        }
        if (kdoKoliduje.tag == "Player")
        {
            Data data = FindObjectOfType<Data>();
            data.SebranychMinci++;
            Destroy(gameObject);
        }
    }

    // Update is called once per frame
    void Update()
    {
        if (transform.position.y < -10)
        {
            transform.position = new Vector3(transform.position.x, 1, transform.position.z);
            GetComponent<Rigidbody>().velocity = Vector3.zero;
            //return;

            ZbyvaSebrat--;
            // zvuk
            Destroy(gameObject);
        }
    }
}
```

Rovina.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Rovina : MonoBehaviour
{
    public int CisloHry = 12345;
    public GameObject PrefabZed;
    public GameObject PrefabMince;
    public GameObject Hrdina;
    public GameObject PrefabTyranosaurus;
    public int PocetTyranosauru = 3;


    // Start is called before the first frame update
    void Start()
    {
        Data data = FindObjectOfType<Data>();
        CisloHry = data.CisloHry;
        PocetTyranosauru = data.PocetTyranosauru;

        Collider collider = GetComponent<Collider>();
        Vector3 velikost = collider.bounds.size;

        // spoleham na to, ze velikost.x==velikost.z
        float minxz = -velikost.x / 2+1;
        float maxxz =  velikost.x / 2-1;
        float kamy = 0.5f;

        GameObject zed;
        zed = Instantiate(PrefabZed, new Vector3(0, kamy, minxz), Quaternion.identity);
        zed.transform.localScale = new Vector3(velikost.x, 1, 1);
        zed.GetComponent<Rigidbody>().mass = 1000;

        zed = Instantiate(PrefabZed, new Vector3(0, kamy, maxxz), Quaternion.identity);
        zed.transform.localScale = new Vector3(velikost.x, 1, 1);
        zed.GetComponent<Rigidbody>().mass = 1000;

        zed = Instantiate(PrefabZed, new Vector3(minxz, kamy, 0), Quaternion.identity);
```

```
zed.transform.localScale = new Vector3(1, 1, velikost.z - 4);
zed.GetComponent<Rigidbody>().mass = 1000;

zed = Instantiate(PrefabZed, new Vector3(maxxz, kamy, 0), Quaternion.identity);
zed.transform.localScale = new Vector3(1, 1, velikost.z - 4);
zed.GetComponent<Rigidbody>().mass = 1000;

Random.InitState(CisloHry);
float K = 10f;

int pocetPrekazek = (int)(0.3 * velikost.x / K * velikost.z / K);
for (int i = 0; i < pocetPrekazek; i++)
{
    float x = Random.Range(minxz+K, maxxz-K);
    float z = Random.Range(minxz + K, maxxz - K);
    float y = (i + 2) * K; // nechame je padat z vysky

    zed = Instantiate(PrefabZed, new Vector3(x, y, z), Quaternion.identity);
    zed.transform.localScale = new Vector3(K, K, K);
}

int pocetMinci = 10;
for (int i = 0; i < pocetMinci; i++)
{
    float x = Random.Range(minxz + 1, maxxz - 1);
    float z = Random.Range(minxz + 1, maxxz - 1);
    float y = (pocetPrekazek + 2) * K + 20 + i;
    //y = 5;

    Instantiate(PrefabMince, new Vector3(x, y, z), Quaternion.identity);
}

float hx = Random.Range(minxz + 1, maxxz - 1);
float hz = Random.Range(minxz + 1, maxxz - 1);
float hy = (pocetPrekazek + 2) * K + 20 + pocetMinci;
Hrdina.transform.position = new Vector3(hx, hy, hz);

// Tyranosaurus:
float minimalniVzdalenostOdHrdiny = 10f;
Vector2 hrdinaXZ = new Vector2(hx, hz);
for (int i = 0; i < PocetTyranosauru; i++)
```

```
        {
            float tx;
            float tz;

            do
            {
                tx = Random.Range(minxz + 1, maxxz - 1);
                tz = Random.Range(minxz + 1, maxxz - 1);
            } while (Vector2.Distance(new Vector2(tx, tz), hrdinaXZ)
                       > minimalniVzdalenostOdHrdiny);

            float ty = hy + 5 + 2 * i;
            Instantiate(PrefabTyranosaurus, new Vector3(tx, ty, tz), Quaternion.identity);
        }
}
```

TlacitkoStart.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TlacitkoStart : MonoBehaviour
{
    public UnityEngine.UI.Button tlacitko;

    // Start is called before the first frame update
    void Start()
    {
        tlacitko.onClick.AddListener(KLIK);
    }

    void KLIK()
    {
        Data data = FindObjectOfType<Data>();
        data.GetComponent<Data>().ZacniNovouHru();
    }
}
```

TlacitkoZpet.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TlacitkoZpet : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        Data data = FindObjectOfType<Data>();
        int skore = 1000 * data.SebranychMinci - (int)data.DelkaHry;

        GameObject vysledek = GameObject.Find("Vysledek");
        vysledek.GetComponent<UnityEngine.UI.Text>().text = skore.ToString();

        GameObject tlacitko = GameObject.Find("Button");
        tlacitko.GetComponent<UnityEngine.UI.Button>().onClick.AddListener(KlikZpet);
    }

    void KlikZpet()
    {
        UnityEngine.SceneManagement.SceneManager.LoadScene("Uvod");
    }
}
```

Tyranosaurus.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Tyranosaurus : MonoBehaviour
{
    GameObject mujHrdina;
    Vector3 kdeJsemNaposledyVidelHrdinu;
    Rigidbody rb;

    float RYCHLOST = 20f;
```

```
// Start is called before the first frame update
void Start()
{
    Data data = GameObject.Find("Data").GetComponent<Data>();
    RYCHLOST = 10f * data.RychlostTyranosaura;

    mujHrdina = GameObject.FindGameObjectWithTag("Player");
    kdeJsemNaposledyVidelHrdinu = Vector3.zero;
    rb = GetComponent<Rigidbody>();
}

bool uzDopadl = false;
void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Rovina")
    {
        uzDopadl = true;
    }
}

// Update is called once per frame
void Update()
{
    if (uzDopadl == false)
    {
        return;
    }

    float BLIZKO = 1f;
    float KROK = 2f;

    // jestli vidim Hrdinu:
    RaycastHit hit;
    Vector3 smer = mujHrdina.transform.position - transform.position;

    if ((Physics.Raycast(transform.position, smer, out hit))
        && (hit.collider.gameObject.tag == "Hrdina"))
    {
        kdeJsemNaposledyVidelHrdinu = mujHrdina.transform.position;
        rb.velocity = Time.deltaTime * RYCHLOST * smer.normalized;
    }
```

```
        else // pokracovat ve smeru, pokud tam jeste nejsem
        if ((transform.position-kdeJsemNaposledyVidelHrdinu).sqrMagnitude>BLIZKO)
        {
            smer = mujHrdina.transform.position - transform.position;
            rb.velocity = Time.deltaTime * RYCHLOST * smer.normalized;
        }
        else // uz jsem tam dosel a Hrdinu nevidim => zvolit si novy cil
        {
            float uhel = Random.Range(0f, Mathf.PI * 2);
            kdeJsemNaposledyVidelHrdinu
                = new Vector3(KROK * Mathf.Sin(uhel), 0, KROK * Mathf.Cos(uhel));
        }
    }
}
```

UNITY
První seznámení s tvorbou počítačových her

Tomáš Holan

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2020
Kniha vyšla jako 26. publikace v Edici CZ.NIC.
Tisk: H.R.G. spol. s r.o., Svitavská 1203, 570 01, Litomyšl
Sazba: Tomáš Brejcha

ISBN 978-80-88168-57-7 (tištěná verze)
ISBN 978-80-88168-58-4 (ve formátu EPUB)
ISBN 978-80-88168-59-1 (ve formátu MOBI)
ISBN 978-80-88168-60-7 (ve formátu PDF)

CZ.NIC | SPRÁVCE DOMÉNY CZ

Edice CZ.NIC

O knize Pokud jste o Unity nikdy neslyšeli, tak je to nástroj, který dovoluje vytvářet profesionálně vypadající 2D i 3D počítačové hry běžící na různých platformách, od Windows přes webové aplikace až po mobilní telefony.

O Unity existuje na internetu taková záplava informací, návodů a tutoriálů, že může být těžké rozhodnout se, jak a čím začít. V této knize se autor snaží na příkladu vytváření jedné hry představit základní principy, techniky a stavební prvky, a zároveň upozornit na možné problémy a jejich řešení.

Kniha nemá formát příručky nebo učebnice, jde spíše o postupné objevování toho, jaké úkoly potřebujeme při vytváření hry řešit a jaké prostředky k tomu Unity poskytuje.

Čtenář by měl mít alespoň minimální povědomí o programování týkající se proměnných, objektů, dosazení, cyklů a funkcí. Zdrojový kód se v Unity píše v jazyku C#, ale k porozumění příkladům bude stačit i znalost nějakého jiného jazyka.

Kniha je určena především pro zvědavé studenty středních škol a možná i pro jejich učitele, kteří hledají nějaké zajímavé téma do kurzů informatiky. Při psaní knihy měl autor na mysli také všechny ostatní čtenáře, kteří o Unity už slyšeli a chystají se s ním již nějakou dobu seznámit, ale zatím nevěděli, odkud začít. Tak teď je ta správná chvíle!

O autorovi **Tomáš Holan** je učitel a jako programátor pracoval na řadě různorodých projektů (assembler i8080, Pascal, Delphi, C#, Flash, PHP, Java), jako byly operační systém pro osmibitové počítače ÁMOS, program na vedení podvojného účetnictví, studijní informační systém, syntaktický analyzátor českých vět, výuková hra o Evropské unii Evropa 2045, hra pro mobilní telefony nebo výuková aplikace pro geometrii GeoTest. Je autorem a spoluautorem několika knih, asi stovky článků a jednoho komiksu.

O edici Edice CZ.NIC je jednou z osvětových aktivit správce české národní domény. Ediční program je zaměřen na vydávání odborných, ale i populárně naučných publikací spojených s Internetem a jeho technologiemi. Kromě tištěných verzí vychází v této edici současně i elektronická podoba knih. Ty je možné najít na stránkách knihy.nic.cz.

knihy.nic.cz

ISBN 978-80-88168-57-7

9 788088 168577