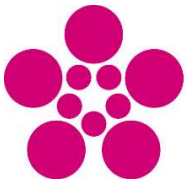


Ekonomická fakulta
Faculty of Economics

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

Jihočeská univerzita v Českých Budějovicích

Ekonomická fakulta

Katedra aplikované matematiky a informatiky

Diplomová práce

# Realistické metody generování modelů krajiny prostřednictvím frameworku Unity

Vypracoval: Bc. Daniel Hejplík

Vedoucí práce: Ing. Jan Fesl, Ph.D.

České Budějovice 2022

**Jihočeské univerza v Českých Budějovicích**
**Přírodovědecká fakulta**

# ZADÁVACÍ PROTOKOL DIPLOMOVÉ PRÁCE

**Student:** Bc. Daniel Hejplík
*(jméno, příjmení, tituly)*

**Obor - zaměření studia:** Aplikovaná Informatika, Softwarové inženýrství

**Katedra:** Ústav aplikované informatiky

**Školitel:** Ing. Jan Fesl, Ph.D.
*(jméno, příjmení, tituly, u externího š. název a adresa pracoviště, telefon, fax, email)*

**Garant z Přírodovědecké fakulty:** .....
*(jméno, příjmení, tituly, katedra - jen v případě externího školitele)*

**Školitel - specialista, konzultant:**
*(jméno, příjmení, tituly, u externího š. název a adresa pracoviště, telefon, fax, email)*

**Téma bakalářské práce:**
Realistické metody generování modelů krajiny prostřednictvím frameworku Unity

**Popis práce:**

Umělé generování realistického vzhledu krajiny je netriviální problém, který vyžaduje pro dosažení co nejlepších výsledků využití sofistikovaných metod. Úkolem práce bude detailně zmapovat tyto metody, popsat je na úrovni matematického aparátu a následně implementovat v aplikaci, která bude schopnosti těchto metod vizuálně demonstrovat.

**Cíle práce:**

1) Provést detailní popisnou rešerši metod pro generování realistického vzhledu krajiny jako jsou metody pro vytváření modelů přírodního terénu prostřednictvím biomů (řeky, hory, údolí), metody evoluce změny terénu prostřednictvím živlů (např. hydraulická a větrná eroze) a metody umožňující dosažení realistického vzhledu (texturace).
2) Detailně matematicky popsat principy zmíněných metod, aby bylo možné tyto implementovat bez nutnosti aditivních knihoven.
3) Popsat možnosti využití frameworku Unity pro generování přírodní krajiny.
4) Implementovat a odladit pilotní verzi aplikace, která bude demonstrovat schopnosti použitých metod v jazyku C# či Python.
5) Provést evaluaci dosažených výsledků a popsat slabé i silné stránky použitých metod.

# Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své diplomové práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne 9. 9. 2022 Podpis studenta

## Poděkování:

Děkuji Ing. Janu Feslovi, Ph.D. za cenná doporučení, věcné připomínky a trpělivost při zpravování mé diplomové práce.

# Obsah

# 1 Úvod

Vytvoření virtuálního prostředí je dnes velmi žádanou činností. Jeho tvorba je v počítačové grafice často spojena s „umělci“, tedy osobou, která ručně vytváří cílený produkt, lze avšak vytvářet i automatizovaně.

Automatizované generování terénu je netriviální problém, pro jehož řešení je nutné vytvoření sofistikovaných metod, poskytujících možnost parametrizace dat pro potřeby cílového prostředí. Na základě těchto dat dochází poté ke zpracování produktu (základního terénu) a jeho vzhledové zlepšení, aby co nejvíce odpovídalo tomu, jak vypadá reálný svět.

K pochopení těchto procesů je potřeba porozumět tomu, jak je v 3D grafice reprezentován prostor a následně i způsobům jeho texturování. K automatizované tvorbě terénu se využívají metody pro generování náhodných hodnot (perlin noise), k následnému zlepšení jeho detailů pak přírodou inspirované procesy (hydraulic errosion).

## 1.1 Cíle

Cílem této práce je detailní zmapování metody generování základního terénu, založené na vytváření modelů přírodního terénu a následné vylepšení detailů na základě přírodními živly inspirovaných metod.

Dále je součástí práce popis fungování frameworku Unity a jeho možnosti pro generování krajiny.

Praktická část práce se zabývá implementací vlastní pilotní verze aplikace, jejíž úkolem je demonstrovat schopnosti metod probraných v teoretické části.

Závěrem je evaluace výsledků a popis silných i slabých stránek použitých postupů.

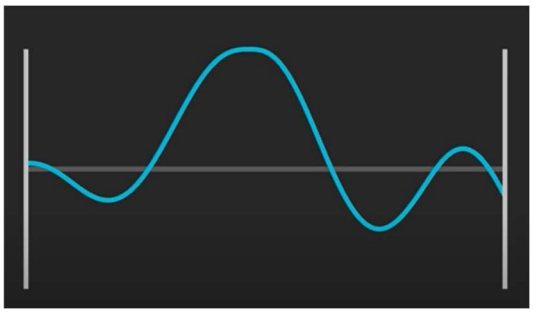
# 2 Metody pro generování krajiny

Terén neboli krajina je 3D objekt reprezentovaný jako výškové rozdíly oproti rovné ploše. V jednoduchém případě by se mohlo jednat pouze o rovný „čtverec“ posunutý o height mapu. Nicméně na této mapě nelze poté dělat optimalizaci jinými metodami. Především je to ale špatný nápad, protože posun height mapou v materiále se počítá při každé změně pohledu na objekt. Z těchto důvodů bude využita tato mapa, ale její posuny budou fyzicky započteny do meshe, nikoliv za pomoci textur.

Na základě vytvořeného povrchu se určí rozdělení popisující typy povrchů v určitých výškách terénu. S těmito daty lze poté vytvořit vzhled výchozího terénu, který se nakonec vylepší pomoci přírodou inspirovaných postupů.

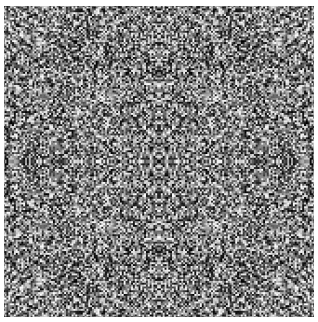
## 2.1 Šum

Generování prostoru se docílí využitím náhody reprezentované jako textura. U této textury určíme jednu z hodnot jako základní výšku a ostatní využijeme jako posun určitým směrem.



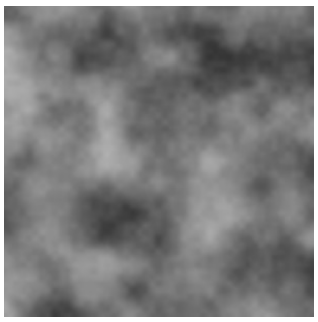
*Obrázek 1 - Změny výšky (vytvořeno pomocí: http://www.maxmcarter.com/sinewave/sinegen.html)*

Nejjednodušším způsobem, jak vytvořit taková data, je vygenerování náhodných hodnot a jejich zaznamenání do textury. Tomuto postupu se říká Normal Noise [1].



*Obrázek 2 - Normal Noise*

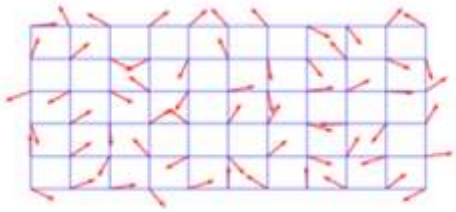
Terén vytvořený na základě této textury by však byl extrémně skokový. Proto je potřeba zajistit v šumu určitou spojitost. Tuto spojitost poskytuje princip zvaný Perlin Noise [2].



*Obrázek 3 - Perlin Noise*

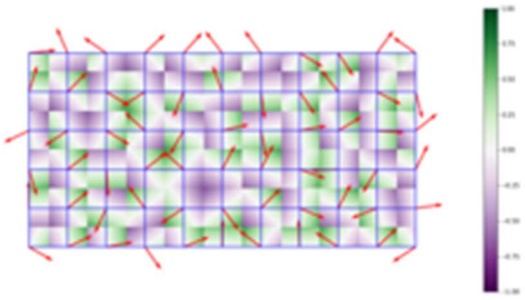
K jeho vytvoření se využívá tří kroků:

1. Navržení mřížky, kde každý průsečík obsahuje hodnotu náhodného vektoru.



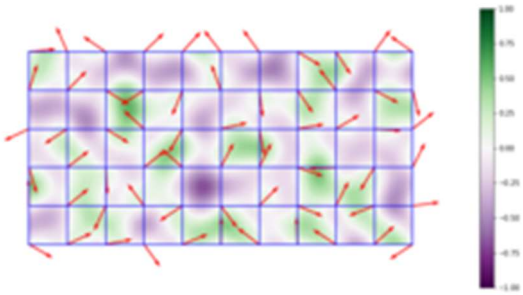
*Obrázek 4 - Noise | Grid*

2. Pro každý bod na textuře nalezneme čtverec v mřížce, na kterém se nachází. Určíme 4 rohové body tohoto čtverce a spočteme pro ně offset hodnotu. Ta se počítá jako rozdíl potřebný k tomu, aby se skutečný rohový vektor dostal na náš hledaný bod. Pro všechny 4 body vytvoříme dot product mezi skutečnou hodnotou vektoru a hodnotou spočteného offset.



*Obrázek 5 - Noise | Dot Product*

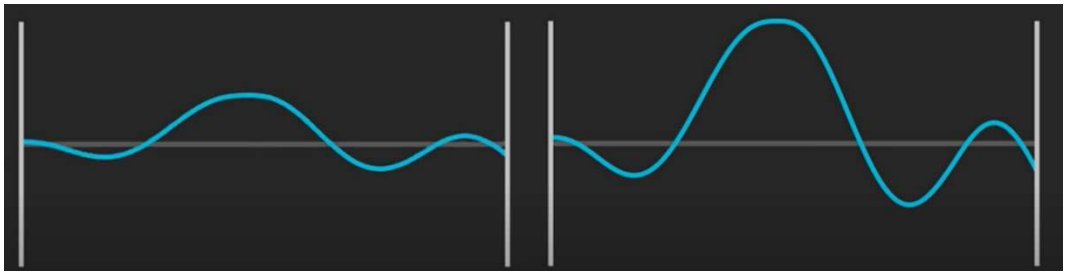
3. Posledním krokem je vytvoření interpolace těchto 4 hodnot. Toho docílíme vypočtením jejich derivace. Pro body, které se nacházejí velmi blízko na mřížce je výsledek derivace aproximací dot produktu původního vektoru. To znamená, že v každém čtverci projde šum hodnotou 0, což vytváří specifický vzhled této textury.



*Obrázek 6 - Noise | Inerpolation*

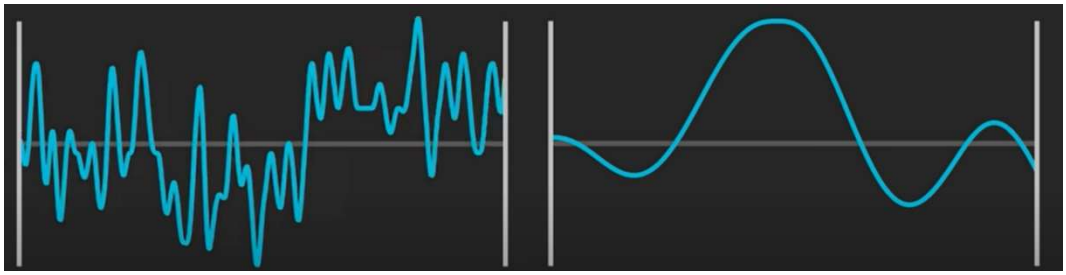
Generování šumu lze brát jako matematickou funkci a této funkci jsme schopni určit amplitudu a frekvenci.

Amplituda určuje výšku dosahovanou extrémními hodnotami. Tedy osu X.



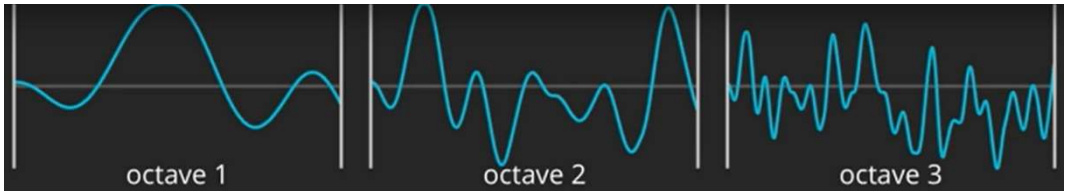
*Obrázek 7 - Amplituda (vytvořeno pomocí: http://www.maxmcarter.com/sinewave/sinegen.html)*

Frekvence popisuje, jak rychle se hodnoty mění na ose Y.



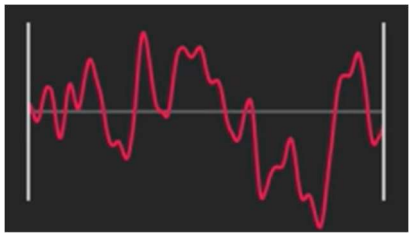
*Obrázek 8 - Frekvence (vytvořeno pomocí: http://www.maxmcarter.com/sinewave/sinegen.html)*

Tato náhodná generace je využitelná, ale je příliš hladká a je tedy potřeba do ní přidat dodatečné detaily propojením několika těchto šumů do jednoho. To však nesmí změnit celkový tvar šumu. Těmto rozšiřujícím frekvencím můžeme říkat oktávy.




*Obrázek 9 - Oktávy (vytvořeno pomocí: http://www.maxmcarter.com/sinewave/sinegen.html)*

Pouhým sečtením oktáv se vytvoří spojitý tvar s větší nahodilostí, avšak ztratí se původní tvar.



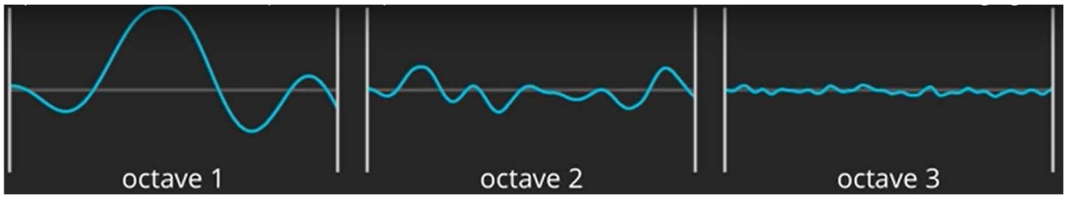
*Obrázek 10 - Finální frekvence (vytvořeno pomocí: http://www.maxmcarter.com/sinewave/sinegen.html)*

Proto je potřeba, aby se se zvětšující frekvencí snížila váha ovlivnění výsledné textury. Pro tento postup lze navrhnout dvě hodnoty, které nám určí, jak se mezi jednotlivými oktávami mění frekvence s amplitudou.

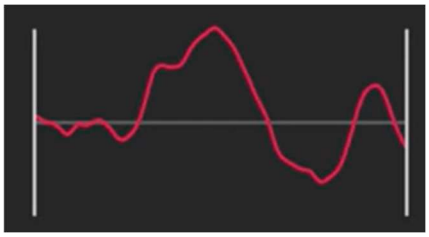
$$Frekvence = x\text{^}octave$$

$$Amplituda = y\text{^}octave$$

Hodnota octave je číslo oktávy, tedy hodnota jdoucí postupně 1,2,3 ...
x je hodnota > 1, což způsobí, že hodnota frekvence s oktávou stoupá.
y je hodnota < 1, proto se amplituda postupně snižuje.




*Obrázek 11 - Lepší oktávy (vytvořeno pomocí: http://www.maxmcarter.com/sinewave/sinegen.html)*



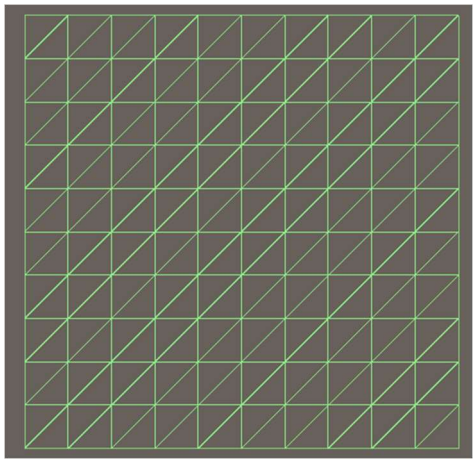
*Obrázek 12 - Lepší finální frekvence (vytvořeno pomocí: http://www.maxmcarter.com/sinewave/sinegen.html)*

Natavení hodnot x určuje množství malých detailů, které jsou ve výsledné textuře. Hodnota y určuje persistenci těchto malých detailů, tedy to, jak moc výsledek ovlivní.

## 2.2 Generování buněk (Mesh generation)

Na základě vytvořené mapy lze generovat fyzický mesh reprezentující potřebný terén. U terénu je jistota kompletní spojitosti celého meshe. Tedy jedná se o „přetvarovanou“ plochu. Proto lze začít se čtvercem rozděleným na menší dle potřebného detailu, kdy počet dělení je potřeba znát dopředu. Díky generovaní z textury můžeme vzít počet pixelů v textuře a využít je jako body rozdělující mesh na menší čtverce.

Vrcholů bude $Výška \times Šířka$ z textury Mesh je reprezentován trojúhelníkovými polygony, proto je potřeba vytvořit $(Výška - 1) \times (Šířka - 1) \times 6$ polygonů.
Výsledkem je rovná plocha o správném počtu polygonů a vrcholů. Dále stačí určit změny na ose Z, kde pozice vrcholů jsou posunuty o hodnotu stanovenou v textuře.



*Obrázek 13 - Mesh*

## 2.3 Tvorba detailů

Krajina vygenerovaná pomocí předchozích kroků reprezentuje tvar terénu. Nicméně je stále tvořena na základě náhodně vygenerovaného koherentního šumu. To nemůže poskytnout detaily, které jsou v reálné přírodě, a proto je potřeba tyto detaily přidat. Vhodným postupem pro generovaní detailů je simulace procesů, díky kterým vznikají v reálném světě. Tyto procesy vypočítávají pohyb, který je tvořen nad původním terénem, a na jeho základě mění výšku různých částí terénu.

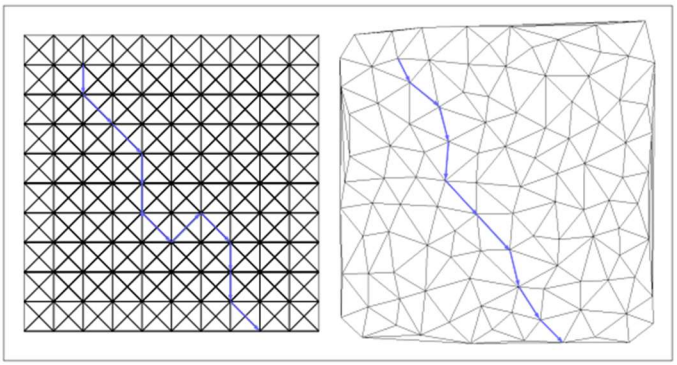
### 2.3.1 Hydraulická eroze

Hydraulická eroze je založena na procesu, kde na místa v terénu dopadá z určitého zdroje kapalina (například déšť). Jednotlivé kapky z tohoto zdroje směřují z vrcholu směrem na nejnižší možnou pozici. Jejich cesta je tvořena cestou nejmenšího odporu. Při cestě tato voda rozpustí část terénu a tyto mikroskopické částečky nese s sebou [3]. Pokud je tohoto sedimentu v kapalině příliš mnoho, tak ho voda již neunese, a proto zůstane přemístěný. Rychlost vody i úhel terénu určují množství sedimentu, který je kapalina schopna nést.
Simulace tohoto procesu je založena na zdrojích kapaliny umístěné na vyšších pozicích v terénu. Z těchto pozic poté simulujeme cestu jednotlivých kapek, kde kapka je reprezentována jako XY souřadnice. Máme jistotu, že kapka je výškově na nejnižším možném místě, proto není potřeba hlídat Z osu. Jelikož se jedná o pouhé kapky, nikoliv

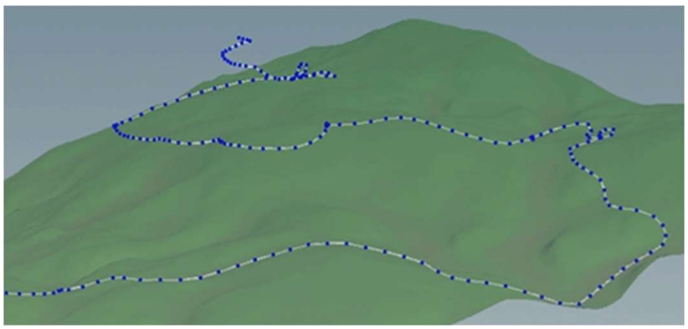
o zdroj stále tekoucí vody, není nutno reprezentovat tyto body vizuálně. Jejich simulace může probíhat pouze datově [4].

Pro každou kapku je vytvořen záznam určující její současný směr pohybu, rychlost a množství vody, kterou reprezentuje. Poslední hodnotou je množství přemisťovaného sedimentu, jehož limity jsou určeny ostatními proměnnými.

Když je kapka vytvořena, potřebujeme simulovat její pohyb směrem k nejnižšímu možnému bodu, kam se dostane cestou s nejmenším odporem.



*Obrázek 14 - 2D cesta kapky eroze*



*Obrázek 15 - 3D cesta kapky eroze*

Pro všechny pozice na mesh terénu jsme schopni určit její výšku, tak i výšku sousedních bodů. Vzniknou nám čtyři výpočty definující gradienty sousedních bodů.

Příklad pro pozici:

$$Pozice_{původní} = (X + a, Y + b)$$

kde [X,Y] jsou nové souřadnice, a hodnoty [a,b] jsou hodnoty [0,1] určující směr sousedů. Vzniknou tím čtyři gradienty:

$$g(Pozice_{x,y}) = \begin{pmatrix} P_{x+a,y} - P_{x,y} \\ P_{x,y+b} - P_{x,y} \end{pmatrix}$$

$$g(Pozice_{x+a,y}) = \begin{pmatrix} P_{x+a,y} - P_{x,y} \\ P_{x+a,y+b} - P_{x+a,y} \end{pmatrix}$$

$$g(Pozice_{x,y+b}) = \begin{pmatrix} P_{x+a,y+b} - P_{x,y+b} \\ P_{x,y+b} - P_{x,y} \end{pmatrix}$$

$$g(Pozice_{x+a,y+b}) = \begin{pmatrix} P_{x+a,y+b} - P_{x,y+b} \\ P_{x+a,y+b} - P_{x+a,y} \end{pmatrix}$$

Provedením interpolace těchto gradientů je vypočten celkový gradient:

$$g(Pozice_{původní}) = \begin{pmatrix} (P_{x+1,y} - P_{x,y}) \times (1-b) + (P_{x+1,y+1} - P_{x,y+1}) \times b \\ (P_{x,y+1} - P_{x,y}) \times (1-a) + (P_{x+1,y+1} - P_{x+1,y}) \times a \end{pmatrix}$$

Tento vektor je poté využit pro vypočtení nového směru pohybu. Šlo by využít pouze tohoto vektoru, ale tím by se pouze zvětšily již existující prohlubně v cestě kapky. Proto je potřeba určit hodnotu setrvačnosti. Ta omezí prudkost, jakou je kapka schopna změnit směr.

$$\text{Směr}_{\text{nový}} = \text{Směr}_{\text{původní}} \times \text{setrvačnost} - \text{g}(\text{Pozice}_{\text{původní}}) \times (1 - \text{setrvačnost})$$

Může nastat situace, kde původní směr a jeho nová změna jsou přesně opačné. V takovém případě jsme se dostali do bodu, na kterém kapka bude do nekonečna rotovat.

Nová pozice se následně vypočte posunutím původní pozice kapky o nově vypočtený směr.

$$\text{Pozice}_{\text{nový}} = \text{Pozice}_{\text{původní}} + \text{Směr}_{\text{nový}}$$

V žádném z předchozích kroků nebyla nijak využita rychlost. Jelikož hodnoty a, b jsou 0 či 1, dochází každým krokem k posunutí přesně o jednu pozici. Kdyby docházelo k posunu na základě rychlosti, mohlo by dojít k „přeskočení“ některých oblastí a tím by postup nemusel být konzistentní. Některé části by mohly být opakovaně ignorovány, a tak by časem vznikly výstupky/prohlubně. Proto je rychlost využita jako parametr určující množství sedimentu, který je kapka schopna přenést. Tím je zachován vliv rychlosti, ale nemůže dojít ke skokovým chybám.

Když víme novou pozici kapky, je potřeba identifikovat materiál, který se mezi danými místy mohl přesunout. Pro obě pozice: $\text{Pozice}_{\text{nový}}$ a $\text{Pozice}_{\text{původní}}$ zjistíme jejich výšky. Na jejich základě poté identifikujeme, zda kapka v posledním kroku stoupala, či klesala.

$$\text{Výška}_{\text{rozdíl}} = \text{Výška}_{\text{nová}} - \text{Výška}_{\text{původní}}$$

Pokud je $\text{Výška}_{\text{rozdíl}}$ pozitivní hodnota, byl poslední pohyb směrem nahoru. Je tedy potřeba „odložit“ přenášený sediment do prohlubně, přes kterou pravděpodobně kapka

právě protekla. Když je $\text{Výška}_{\text{rozdíl}}$ negativní hodnota, jedná se o cestu směrem dolů. V tuto chvíli může kapka „odebrat“ sediment z původní pozice, aby ho přemístila jinam. Množství odebraného sedimentu je omezeno limitní kapacitou, kterou je kapka schopna pojmout. Kapacitu určíme na základě rychlosti a úhlu po klesání. Nicméně pokud by se blížil 0, blížila by se 0 i kapacita. Tento postup by způsobil, že rovné plochy by si pouze posouvaly přebytky o jednu pozici vedle a tím by se postupně vytvářely skokové vrcholy na místech, kde kapka rychle ztratila rychlost. Z těchto důvodů je úhel limitován o minimální hodnotu, která zařídí, že prudké ztráty rychlosti nejsou tak extrémní.

$$\text{kapacita} = \max(-\text{Výška}_{\text{rozdíl}}, \text{minimální úhel}) \times \text{rychlost} \times \text{velikost kapky}$$

Pokud je přenášeno větší množství sedimentu, než je povolená kapacita, dojde k „odložení“ přebytků na základě hodnoty určující „shlukování“ sedimentu. Tato hodnota změní chování způsobu navazování jednotlivých kapek.

$$\text{sediment}_{\text{odložený}} = (\text{sediment}_{\text{nesený}} - \text{kapacita}) \times \text{shlukování}$$

Při hodnotě shlukování 1 dochází k situaci, kdy je odložen veškerý přebytečný sediment. Pokud však chceme vytvořit hladší variaci terénu a tím simulovat větší množství kapek, které nepovolují celému sedimentu se oddělit, lze shlukování nastavit na hodnotu mezi 0 a 1.

Když je přenášeno menší množství, než je kapacita, algoritmus naopak odebere materiál z původní pozice, a přidá ho do svého neseného sedimentu. Je nutné zařídit, aby nikdy nebyl odebráno více materiálnu, než je rozdíl mezi dvěma ověřovanými pozicemi. V situaci, kdy by kapacita mohla přijmout více materiálu, než je rozdíl a nebylo to nijak ohlídáno, docházelo by k tvorbě vykopaných děr na místech, kam kapka dorazila s velkou volnou kapacitou.

$$\text{sediment}_{\text{nesený}} = \min((\text{kapacita} - \text{sediment}_{\text{původní}}) \times \text{eroze}, -\text{Výška}_{\text{rozdíl}})$$

V momentě, kdy víme kudy se kapka bude pohybovat a jakým způsobem přemístí sediment, je potřeba vypočítat novou rychlost. Tu lze určit přidáním $\text{Výška}_{\text{rozdíl}}$ násobené silou gravitace. Tento postup však nijak nepočítá s hodnotou setrvačnosti. Pokud zdůrazníme původní rychlost tak, aby její vliv byl větší než rozdíl výšek, dojde k lepšímu zachování setrvačnosti.

$$\text{rychlost}_{\text{nová}} = \sqrt{\text{rychlost}_{\text{původní}}{}^2 + \text{výška}_{\text{rozdíl}} \times \text{gravitace}}$$

Jelikož s postupem cestování se části kapky zachycují na odložených sedimentech, či se vypaří, je potřeba zajistit změnu velikosti kapky. Tato změna nám současně zařídí ukončení celého procesu tak, aby se kapka ve finální oblasti netočila do nekonečna. Proto vytvoříme hodnotu reprezentující rychlost, s jakou kapka mizí. Hodnotu lze brát jako určení počtu kroků kolik kapka vydrží.

$$\text{velikost kapky}_{\text{nová}} = \text{velikost kapky}_{\text{plvodní}} \times (1 - \text{vypařování})$$

Kombinace těchto kroků získává a přemisťuje výšku částí terénu. Mají však stále jeden problém. Kapky jsou vždy velikosti jednoho bodu, a proto by docházelo k vygenerování jedné specifické cesty pro jeden zdroj. Muselo by dojít k vygenerování tak velkého množství zdrojů, aby se zajistilo pokrytí všech sousedících bodů. To již ale neodpovídá realitě, a hlavně by se časem vyrovnal terén do perfektně rovné plochy. Proto je potřeba zařídit „spadnutí“ materiálu ze sousedních bodů terénu. I v realitě, pokud voda odnese část sedimentu na jednom místě, materiál, který s ním sousedí, nezůstane bez změn stát, ale je stažen do nižšího místa. K nasimulování tohoto procesu si určíme velikost kruhu, kolik sousedních bodů může erodovat společně s odebíráním sedimentu. Tím se částečně rozšíří vliv kapky.
Z popisu je pochopitelné, že chování eroze je dáno sérií parametrů:

- setrvačnost
- vypařování
- velikost kapky
- gravitace
- eroze
- minimální úhel
- shlukování

Vizuální rozdíly vytvořené různými hodnotami těchto parametrů jsou lépe zobrazeny v praktické části práce.

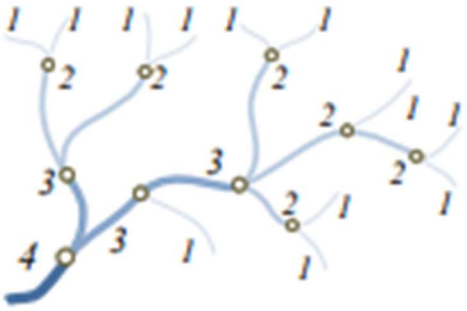
### 2.3.2 Doplnění detailů

Generování pomocí eroze vytvoří dostatečné detaily, ale nastane u ní problém při velkém počtu kroků, nebo je zcela nepoužitelná na terénu s nedostatečným počtem detailů na začátku procesu. V takových případech tím dochází pouze k vyhlazení terénu. Je proto

vhodné i využití metod, které dodatečný detail dokáží doplnit. K tomu je v práci využito dvou postupů.

#### 2.3.2.1 Řeky

Prvním postupem je generování nové plochy založené na chování řek. Podobný postup byl již velmi dobře popsán v Terrain Generation Using Procedural Models Based on Hydrology [5]. Tato studie generuje kompletní simulaci prostoru založenou na generování stromových struktur simulující chování řek.



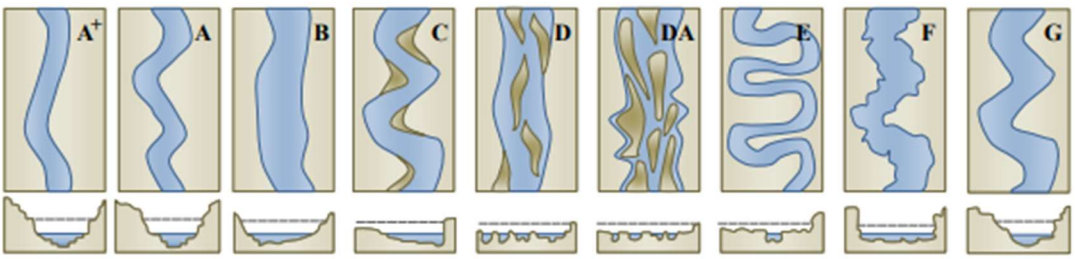
*Obrázek 16 – Stromová cesta Horton-Strahler*

Vrcholy této cesty jsou indexovány dle Horton-Strahlerova čísla, kde každým dělením je číslo snižováno až k finálním listům označeným 1. Jelikož se jedná o rozdělování na dvě části v každém dělícím bodě stromu lze počet vrcholů v každé hloubce stanovit pomocí

$$\frac{n_i}{n_{i+1}},$$

kde $n_i$ určuje počet uzlů v i-tém řádu.

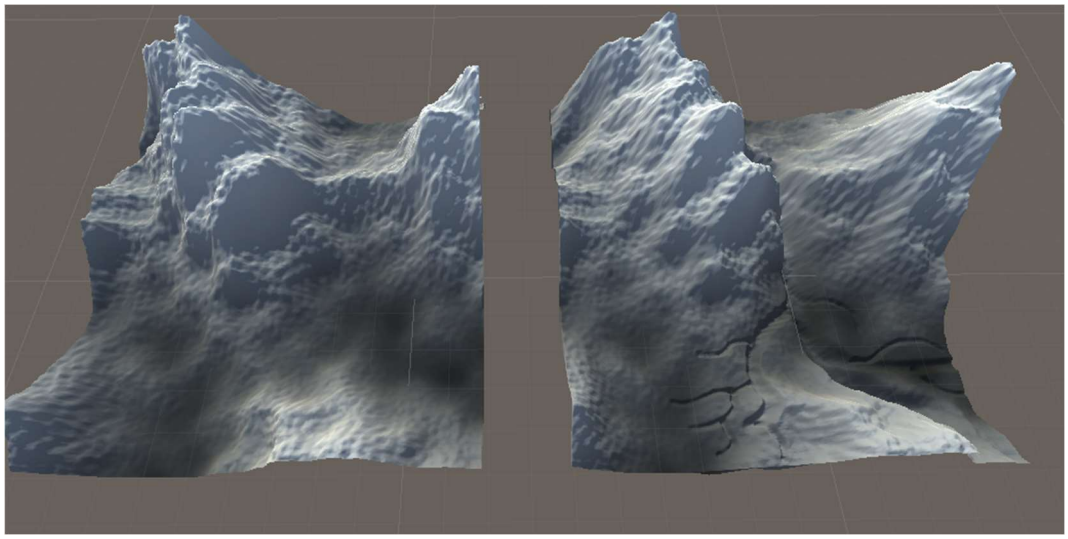
Systém vytváří opakovaně tuto strukturu až do obsazení celé vstupní textury a poté na ní generuje plochy vod dle pravidel určující, jaké typy vodních toků se zde mohou nacházet.




*Obrázek 17 - Typy vodních toků (zdroj: https://hal.archives-ouvertes.fr/hal-01339224/document)*

V našem případě se ale nejedná o generování terénu „ve vakuu“. Je proto nutno postup pozměnit tak, aby mohl být využitelný na již existujícím prostoru. Toho bylo dosaženo změnou pravidla určujícího výchozí bod stromu tak, aby byl na pozici vyšší v původní textuře. Následující body se poté musí nacházet na pozicích nižších.

Tím vzniká série úseček, u nichž můžeme určit šířku a hloubku změny terénu. Jednotlivé úsečky lze považovat za hrany v neorientovaném grafu. Tím můžeme postupně snižovat hodnotu hrany v různých jejich místech. Tzn. vrchol má přesnou hodnotu své výšky a jednotlivé body na této úsečce se postupně snižujíc až do hodnoty následujícího vrcholu.



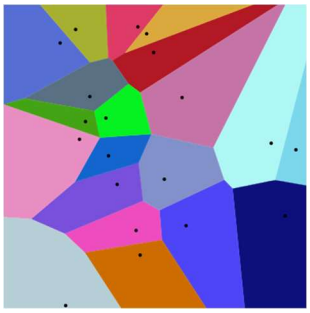
*Obrázek 18 - Doplnění řeky do terénu*

Tato technika je schopná doplnit nové detaily, nicméně stejně jako hydraulická eroze pouze „odebírá“ materiál a snižuje jeho výšku.

#### 2.3.2.2 Tektonické desky

Druhý postup, který byl využit na doplnění nového detailu, je založen na simulaci pohybu tektonických desek.

Pro tuto simulaci je potřeba v terénu uměle vytvořit zóny reprezentující jednotlivé tektonické desky. K tomu lze využít různých postupů. V této práci byl zvolen postup založený na rozdělení plochy do regionů dle vybraných startovních bodů neboli Voronoi diagram [6]. Vychází z postupu, kde se k setu vybraných bodů doplní okolní body a ty se postupně připojují k nějakému regionu. Regiony začínají jako vybrané body a jsou postupně rozšiřovány dle zadaných pravidel tak, aby postupně mezi sebe rozdělily všechny body původního obrazu. Pravidlo určené pro propojení skupiny a bodu může být například euklidovská či manhattanská vzdálenost.



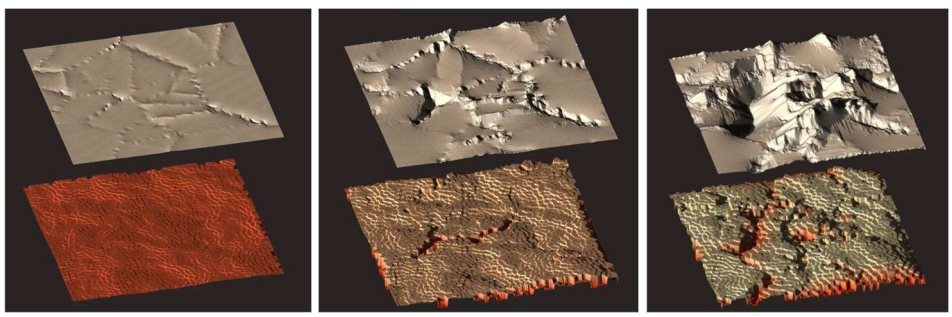
*Obrázek 19 - Voronoi diagram*

Na základě této textury můžeme vytvořit simulaci pohybu desek.

Prvním krokem je výběr desek a jejich pohybu. Na základě této informace poté spočteme překrytí, jeho velikost a pozici. Pro začátek lze určit, že všechny desky, které mají společnou hranu, mají na těchto společných bodech vzdálenost 0. Poté se určí pohyb celé desky (či několika desek) a vypočítá se nová vzdálenost těchto bodů.

Když je zjištěna informace o posunu, je nutno určit horizontální pozici desek. Při sražení desek zvedáme pozice na horní desce, a při vzniku prostoru mezi deskami (tedy překrytí, které je <0) naopak dolní desku snižujeme. Výchozí pozici desek lze určit náhodným indexováním jednotlivých ploch Voronoi textury. Po prvním kroku však již lze využít jejich „skutečné“ pozice. Na základě spočteného pohybu ke každé desce určíme její novou horizontální pozici.

Po X krocích této kolize máme pro terén sérii (-+) hodnot určujících vzdálenost, o kterou se posune původní terén. Pro každý bod na textuře se zjistí horizontální pozice odpovídající tektonické desky. Bod posuneme o hodnotu určenou nejen deskou ke, které spadá, ale ovlivněnou i o pozici X nejbližších desek. Pro ulehčení výpočetních dat jsme deskami posouvali jako celkem, máme pro ni tedy jen jednu informaci. Zjištěním rozdílu oproti dalším nejbližším deskám pro daný bod docílíme vytvoření postupně se měnících hodnot pro jednotlivé body na desce.



*Obrázek 20 - Terén tektonických desek (zdroj: https://nickmcd.me/wp-content/uploads/2021/04/output3.mp4)*

Tento postup může doplnit nové informace jak směrem dolů, tak směrem nahoru.

# 3 Unity

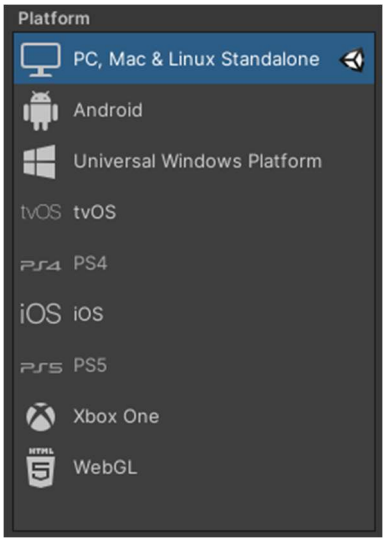
Engine Unity je multiplatformní systém používaný především pro tvorbu her, jeho využití je však vhodné i pro simulační a vizualizační nástroje. Unity obsahuje předpřipravené nástroje, které pomáhají při vývoji. Výsledkem práce v Unity může být jeden ze dvou produktů:

a) Engine Asset – nástroj používaný přímo v Unity, jedná se o rozšíření základních nástrojů o složitější funkcionalitu,
b) Standalone product – vygenerovaná runtime (.exe) aplikace, která v sobě obsahuje uzavřenou funkcionalitu.

Samotný Editor Unity se skládá ze série modulů. [7]

## 3.1 Scripting

Základem kódu v Unity je open-source .NET platforma, je tedy multiplatformní, tj. může fungovat na větším množství zařízení.




*Obrázek 21 - Unity Platformy*

Unity obsahuje dva skriptovací backendy [8]:

a) Mono – využívající just-in-time kompilaci, který kompiluje kód dle potřeby během běhu programu,
b) IL2CPP – využívající ahead-of-time kompilaci, která přeloží celou aplikace před jejím spuštěním.

Při práci v rámci editoru Unity je automaticky využito varianty Mono. Je to z důvodu rychlejší kompilace u JIT založených systémů a také z důvodu nezávislosti na platformě.

Při vytváření finálního buildu produktu je však možné využít variantu IL2CPP, kde je výsledný produkt lépe optimalizovaný, ale vygenerovaný je pouze na určitou platformu.
.NET v Unity není standardním .NET vytvářeným .NET Foundation, ale jedná se o speciální subset rozšířený o vlastní Unity specifické třídy [9].
Při tvorbě projektu lze zvolit jednu ze dvou možností, na kterých je tento subset založen:

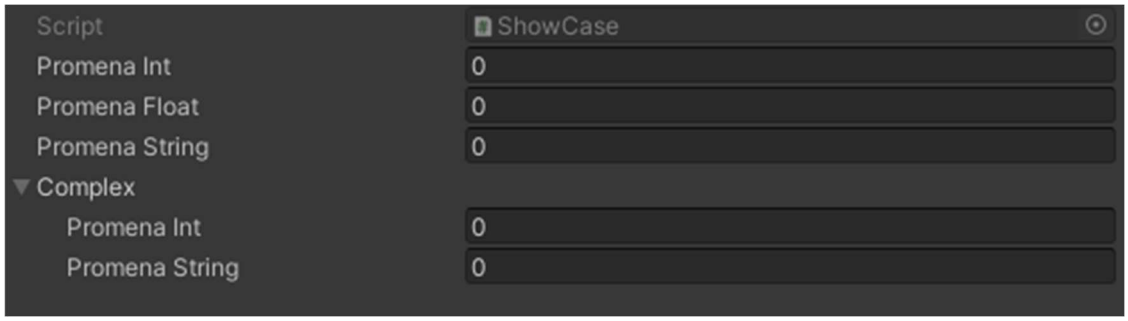
a) .NET Standard 2.0 – je vhodný díky kompatibilitě s naprostou většinou knihoven v .NET,
b) .NET 4.x – Unity automaticky kompiluje proti specifickému profilu .NET Foundation odpovídajícímu jedné z verzí .NET 4 (.NET 4.5, .NET 4.6, .NET 4.7, atd.).

Kvůli větší kompatibilitě se doporučuje využít .NET Standard 2.0, pokud není opravdu nutné využití nějaké funkcionality, která je dostupná pouze v .NET 4.x
.NET, a tedy i všechny scripty v Unity jsou psány v jazyce C#. Ke kompilaci je využito překladače Roslyn a momentálně nejvyšší podporovaná verze jazyka je C# 8.0.
Součástí Unity je také automatizovaná serializace [10]. Ta pomáhá transformovat datové struktury nebo objekty do formátu, který umí Unity rozpoznat a především ukládat, aby je mohl rekonstruovat a pracovat s nimi. Data serializovaného objektu je možno automatizovaně přeložit do formátu, který obsahuje pouze data (ve formátech JSON nebo bitově). Tato data pak lze snadno uchovat v souboru i mimo běh programu.
Unity má bohužel jednu velkou limitaci při serializovaní a tou je nemožnost polymorfismu. Toto omezení se však týká pouze vlastních tříd, lze to tedy obejít využitím některých z `Unity.Object` potomků, kteří jsou na tyto činnosti určeni.
Všechny serializované proměnné jsou zobrazovány v Inspektor okně tohoto objektu.
Výjimkou jsou proměnné označené atributem `[HideInInspector]`, k jejichž serializaci dojde normálně, jen nejsou zobrazeny v Inspektoru.

```
public class ShowCase : ScriptableObject
{
    public int promenaInt;
    [SerializeField]
    private float promenaFloat;
    public float promenaString;
    public ComplexSerialization complex;

}
```

```
[Serializable]
Počet odkazů: 1
public class ComplexSerialization
{
    public int promenaInt;
    [HideInInspector]
    public float promenaFloat;
    public float promenaString;
}
```

*Obrázek 22 - Ukázka serializace*




*Obrázek 23 - Zobrazení serializaci*

### 3.1.1 MonoBehaviour

MonoBehaviour je jednou z nejvyužívanějších tříd [11]. Tato třída reprezentuje chování jedné komponenty, tudíž automaticky poskytuje možnost napojení na určitý herní objekt. Jedinou podmínkou fungování je, aby se soubor, ve kterém se MonoBehaviour nachází, jmenoval přesně stejně jako ona třída *(ShowCase.cs)*

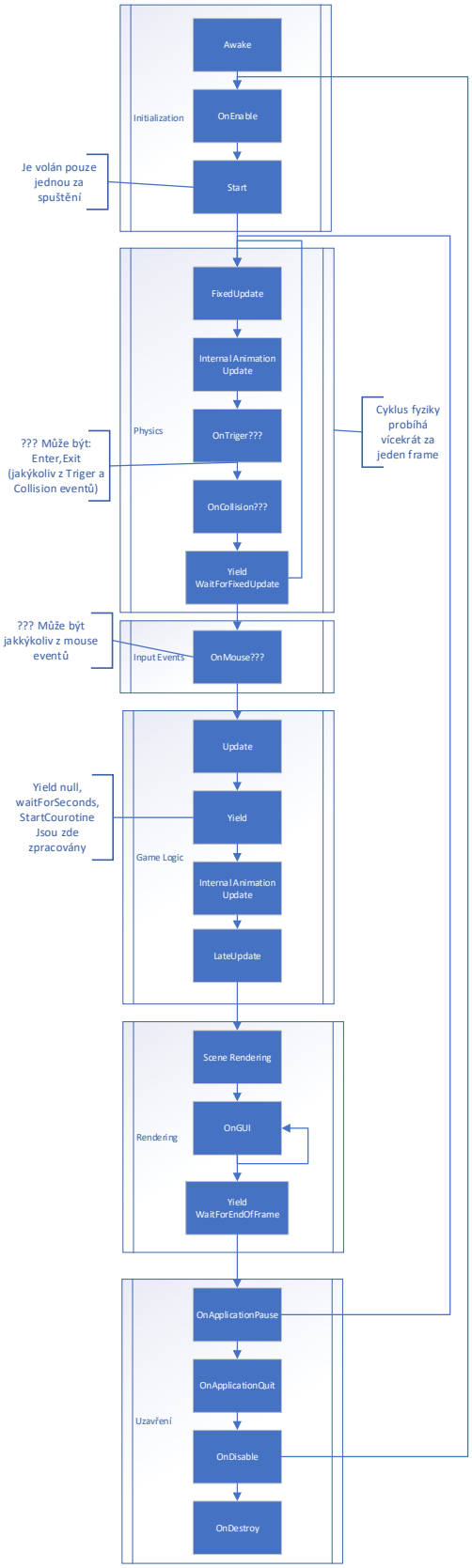
```
using UnityEngine;

Skript Unity | Počet odkazů: 0
public class ShowCase : MonoBehaviour
{
    // Start is called before the first frame update
    Zpráva Unity | Počet odkazů: 0
    void Start()
    {

    }

    // Update is called once per frame
    Zpráva Unity | Počet odkazů: 0
    void Update()
    {

    }
}
```

*Obrázek 24 - MonoBehaviour*

Scripty, které dědí z MonoBehaviour obsahují několik automaticky zpracovávaných metod. Tyto metody jsou volány při běhu programu a tím provádějí logiku komponenty.

#### 3.1.1.1 Cyklus programu

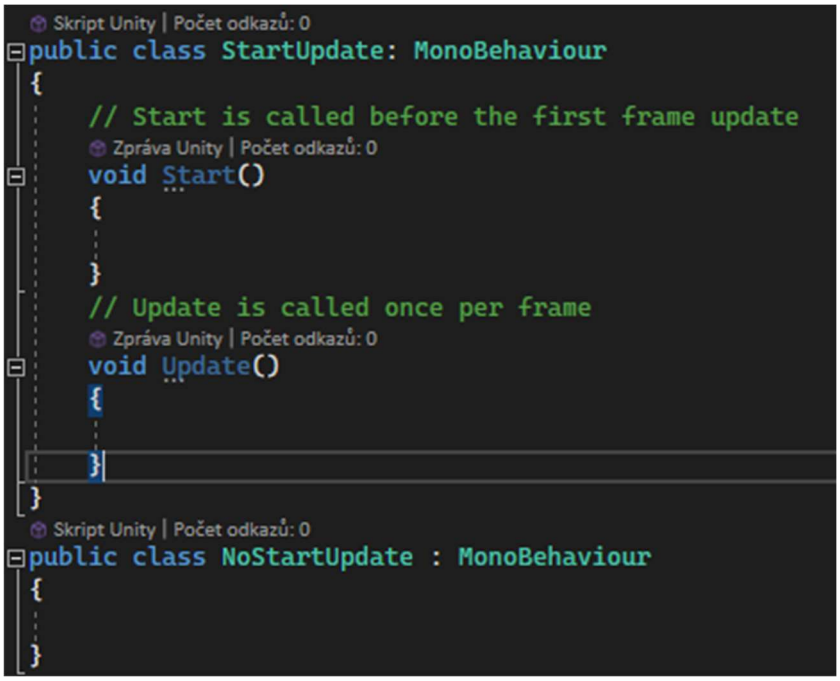
Unity využívá přesně určeného pořadí činností [12].




*Obrázek 25 - Cyklus běhu Unity*

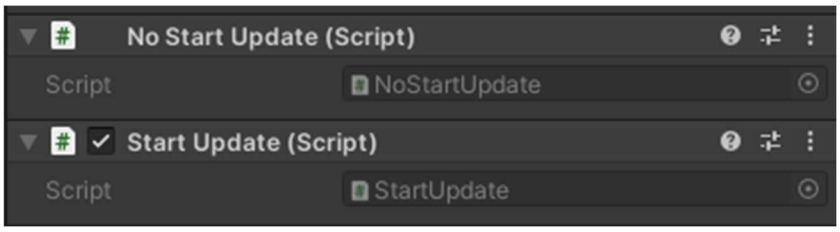
V momentě zapnutí programu je na všech komponentách spuštěna metoda Awake. Pokud je objekt vytvořen až v průběhu programu, jeho MonoBehaviora mají tuto metodu spuštěnou hned po jejich vytvoření.

Některé komponenty mohou být využity jinými a sami nemusí obsahovat zpracování žádné z metod, které tento cyklus využívají.




*Obrázek 26 - Cyklus metody*

Pokud žádná z těchto metod není v třídě obsažena, je tato třída považována za „non-active“ a je vynechávána z cyklu zpracování. Pokud obsahuje alespoň jednu z nich, pak je zpracována pro všechny. V takovém případě přibude v Inspektor okně dané třídy možnost zaškrtnutí, zda je MonoBehaviour aktivní na startu či nikoliv. Při běhu programu jde samozřejmě tento stav kódově změnit.




*Obrázek 27 - Acitve vs non-active*

OnEnable metoda je pak volána při aktivování komponenty, nebo pokud byla aktivní od začátku, tak je volána ihned po metodě Awake.

Metoda Start je volána po OnEnable, ale pouze poprvé. Je zde z důvodů propojování dat, tedy pokud jsou dvě třídy na sobě závislé navzájem, nemáme jistotu, která z nich se načte první. Proto můžeme závislé činnosti odsunout do Start a tím zajistit, že již proběhlo OnEnable u obou z nich.
Další skupinou metod jsou metody Update:

- FixedUpdate
- Update
- LateUpdate

FixedUpdate je prováděn vždy jako první z nich a zpracovává změny vypočtené na základě systému fyziky. Může být proveden i několikrát během jednoho zobrazovaného snímku, neměl by tedy obsahovat žádné náročné výpočty a mělo by se jednat pouze o fyzikální datové změny. Je volán na základě pravidelného časovače, a tak u něho není potřeba hlídat „časově“ závislé změny [13].
Update je hlavní místo, kde se určuje opakovaná činnost, která se v systému děje. Tato metoda je volána právě jedenkrát za snímek. Protože nevíme počet snímků, které jsou v jedné vteřině zpracovány (může se měnit), neexistuje jistota, jak často je tato metoda volána. Z tohoto důvodu je poskytován záznam Time.deltaTime, reprezentující čas, který uplynul od posledního Update. Ten můžeme využít jako hodnotu pro násobení změn, které by měly odpovídat nějakému časovému úseku [14].
LateUpdate je také volán pouze jednou, ale je volán až po zpracování záznamů změn. Příkladem může být pohnutí kamerou, která sleduje pozici několika objektů, jejichž změna proběhla v Update. Tím se zajistí, že všechny změny byly vypočteny před tím než kamera začne počítat své změny.
Mezi FixedUpdate a Update probíhá zpracování fyzických eventů, které Unity na pozadí automaticky zpracovává za pomocí systému PhysX. Během těchto kroků je aktivována skupina metod, které zpracovávají kolize objektů. Také zde proběhne zpracování inputů, tedy zaznamenání zmáčknutých kláves a pozice myši. Tato data lze poté využít v Update/LateUpdate.
Po zpracování Update následuje vytvoření 3D renderu a po něm vytvoření GUI komponent.
V různých fázích cyklu jsou vyvolávány Yield volání, které lze využít k více vláknovému zpracování programu za pomoci Courotine [15]. Tato volání automaticky pracují na vlastním vláknu a pro synchronizaci využívají právě await na metodu obsahující Yield.

Na závěru cyklu je volána série uzavíracích metod. Ty jsou všechny postupně volány, ale provedeny pouze pokud je splněna jejich podmínka.

- OnApplicationPause – pouze když byla během frame aplikace pozastavena,
- OnApplicationQuit – pouze když byla aplikace ukončena,
- OnDisable – pokud bylo toto MonoBehaviour nebo GameObject, na kterém se nachází „vypnut“,
- OnDestroy – volán při zničení objektu (nebo MonoBehaviour komponenty), měl by obsahovat metody pro vyčistění paměti, tedy odstranit data z listů, kde se komponenta nacházela.

### 3.1.2 ScriptableObject

Pro data nezávislá na specifických objektech, tedy bez potřeb propojení s nějakou instancí třídy, existuje specializovaný objekt fungující jako jednoduchý kontejner na data [16].

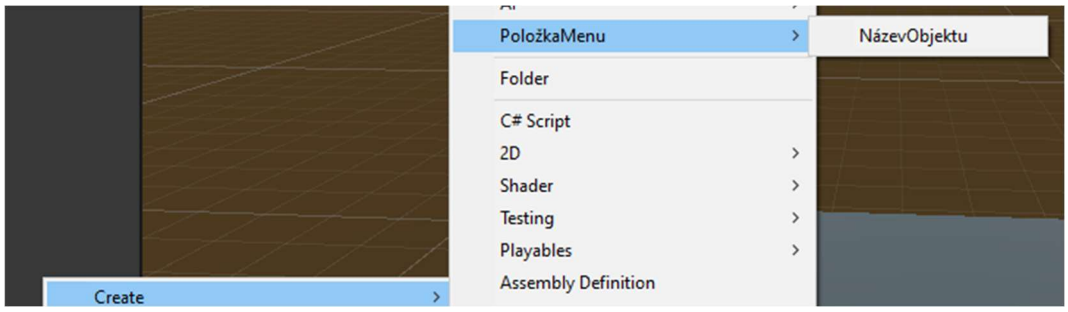
```
[CreateAssetMenu(fileName = "Výchozí název souboru", menuName = "PoložkaMenu/NázevObjektu", order = 1)]
Skript Unity | Počet odkazů: 0
public class ShowCase : ScriptableObject
{

}
```

*Obrázek 28 - ScriptableObject*

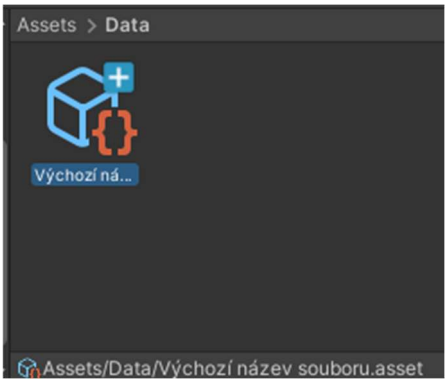
Takto vytvořený skript se nedá spojit s herním objektem, tudíž není spojen s žádnou automatizovanou činností a je pouhými daty. Nevyžívá cyklus popsaný v předchozí kapitole a nelze z něj ani vytvořit standartní objekt pomoci konstruktoru (`new ShowCase()`).

Objekt může vzniknout pouze dvěma způsoby:

- runtime vytvoření pomoci `ScriptableObject.Instantiate<ShowCase>()`,
- vytvoření v Projekt okně – přidáním atributu `CreateAssetMenu` je vytvořeno menu dosažitelné pomocí kliknutí pravého tlačítka v okně Projekt.

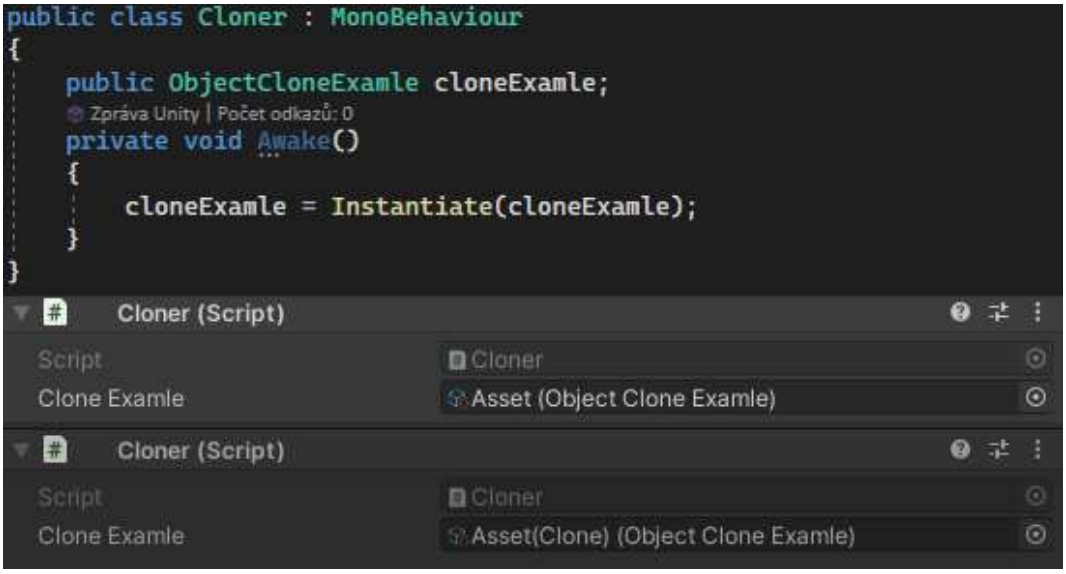



*Obrázek 29 - Vytvoření ScriptableObject*




*Obrázek 30 - Vytvořený ScriptableObject*

Tento objekt je pevně nastaven a není nijak vázán na běh programu. To znamená, že každá změna nad tímto datovým souborem je trvale zachována. Lze tím docílit snadného uchovaní stavu i po vypnutí programu, avšak je nutné hlídat, aby nedošlo ke změnám nad daty, které mají fungovat jako „natavení“ výchozího stavu při spuštění. Tento způsob využití je také vhodný, ale pokud chceme, aby se při startu programu nastavil výchozí stav a ten jsme v mohli v průběhu měnit a současně po vypnutí a zapnutí byl opět v původním stavu, je možné udělat klon původního souboru v Awake metodě MonoBehaviour, který má tyto změny zpracovávat.




*Obrázek 31 - Vytvoření klonu SO*

### 3.1.3 Editor

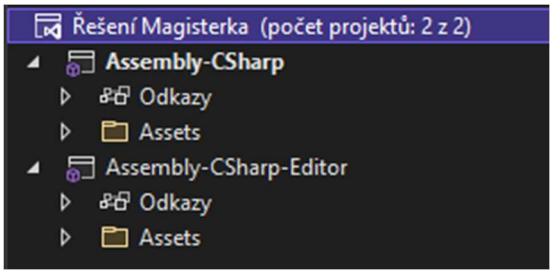
V Unity lze vytvářet vlastní editační okna, nebo upravovat vzhled automaticky generovaných Inspektorů.

Výsledkem těchto úprav jsou nástroje, které nemají žádný vliv na výsledný „produkt“ vygenerovaný z Unity, ale mohou velmi ulehčit práci v editoru. Jelikož jejich kód

zasahuje přímo do samotného Unity editoru, který není součástí sestaveného produktu, není možné, aby byl jejich kód přímo součástí ostatních scriptů [17].
Z těchto důvodů Unity poskytuje dva základní namespace:

- UntiyEngine – standartní třídy a metody, které budou ve výsledku,
- UnityEditor – obsahuje věci zasahující do editoru, není součástí buildu.

Unity automaticky zařazuje všechny scripty, které jsou v projektu vytvořeny pod jeden z těchto prostorů. Dociluje toho pomocí čtení cesty souboru a pokud někde ve své hierarchii obsahuje složku Editor, tak je přeřazen do druhého projektu.



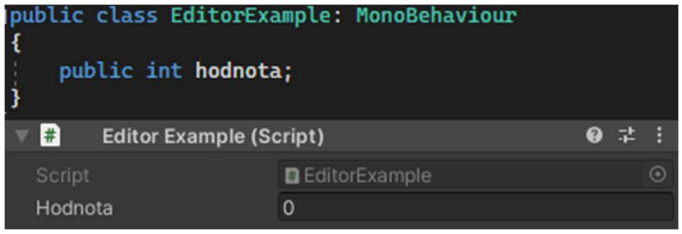

*Obrázek 32 - Engine/Editor Projekt*

V Editor projektu je možné vidět celý kód, ale Engine projekt vidí pouze svůj obsah. Díky tomu může Editor zasáhnout do činností Engine, ale při sestavování je jistota, že v Engine části jsou pouze ty scripty, které opravdu lze sestavit bez závislostí na UnityEditor.
K tvorbě vlastních nástrojů lze použít čtyři přístupy:

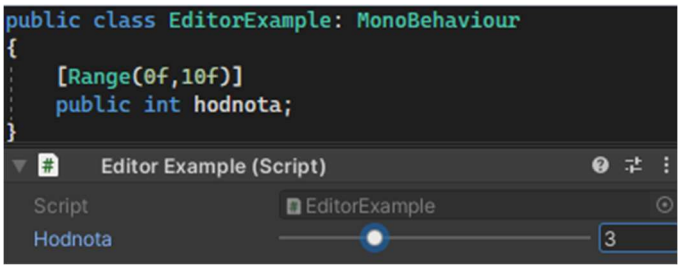
- Atribut Drawer
- Property Drawer
- Inspector
- Editor Window

#### 3.1.3.1 Atribut Drawer

Nejjednodušší metodou, jak upravit vzhled a chování editoru je využití Atributu [18], pro který je vytvořen vlastní styl zobrazení. Unity obsahuje velké množství těchto atributů již od základu. Příkladem může být omezení rozsahu pro číselné proměnné.



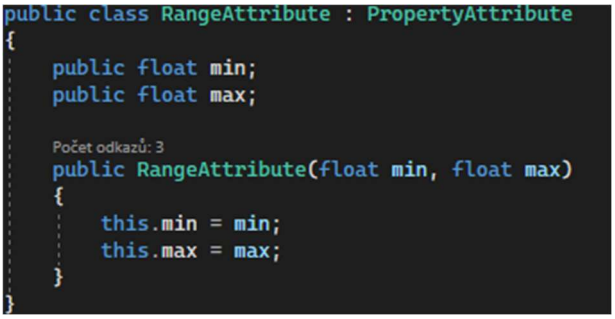

*Obrázek 33 - Bez Range Atributu*




*Obrázek 34 - S Range Atributem*

Lze samozřejmě vytvořit vlastní atributy, k tomu je potřeba dvou kroků.

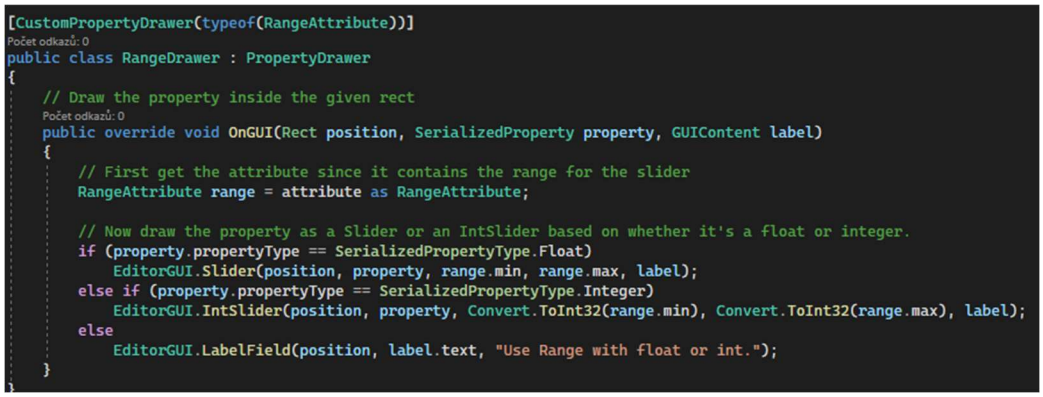
Za prvé je potřeba vytvoření samotného atributu. Ten je vytvořen jako třída dědící z PropertyAttribute a obsahuje pouze data, která určují chování či vzhled, nebo výpočty, které s těmito daty souvisejí.




*Obrázek 35 - RangeAttribute*

Za druhé je potřeba vytvoření samotného draweru, který určí, jak budou data atributu zobrazována.

Přidáním `UnityEditor`, namespace dostaneme přístup k třídám a metodám pro generování vzhledu.



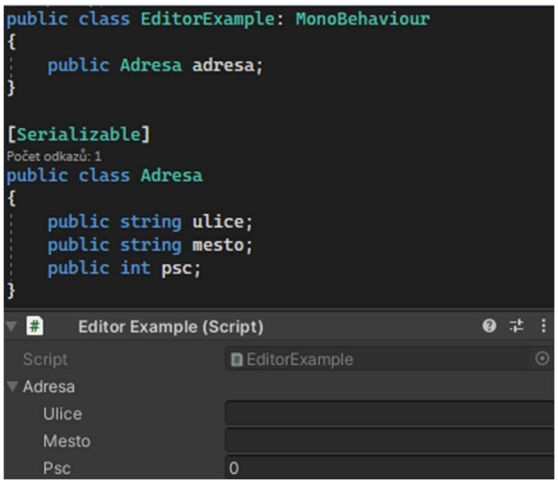

*Obrázek 36 - RangeDrawer*

Drawer je jakákoliv třída nacházející se ve složce Editor, která dědí z PropertyDrawer [19].

`[CustomPropertyDrawer(typeof(RangeAttribute))]` určí, jakému atributu tato třída mění vzhled. V ní stačí overridovat metodu OnGUI, ve které se pomoci statické třídy EditorGUI využijí grafické komponenty poskytované Unity. V příkladu Range zde vidíme ověření, zda se jedná o Float či Integer, a dle toho vytvoříme odpovídající Slider komponentu.

#### 3.1.3.2 Property Drawer

CustomPropertyDrawer může být využit i přímo nad vlastní třídou.




*Obrázek 37 - Příklad serializované třídy bez PropertyDrawer*

Postup je zde sejný jako u atributů, cílem je nastavení vzhledu pro třídu a všechny proměnné v ní. Jak u Atributů, tak u Property je využito PropertyDraweru, avšak u atributů se vždy jedná o zobrazení nad jednou proměnou, zde může dojít k změnám nad více proměnnými. Díky tomu může nastat jeden specifický problém.

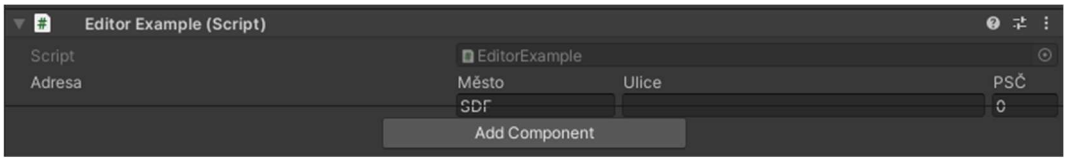
Jako příklad si vytvoříme třídu a chceme, aby se všechny jeho proměnné objevily jako jeden řádek s labely nad nimi. Pokud měníme vzhled pro komplexní třídu a u její proměnné chceme zanechat jejich výchozí vzhled, lze z EditorGUI využít metodu PropertyField, která si automaticky zjistí typ a jeho zobrazení. Samotnou proměnou pak lze získat ze SerializedProperty, která je automaticky předána metodě OnGui.

Tak lze pouhým určením pozice a velikosti pomocí Rect a jeho využití v PropertyField vytvořit zobrazení proměnné. V této metodě se určuje styl zobrazení výchozího labelu. Zde jsme se rozhodli využít bez zobrazení, jelikož pro ukázku chceme adresu ukázat jako jeden řádek s labelem nad políčky.

```
EditorGUI.PropertyField(mestoRect,
property.FindPropertyRelative("mesto"), GUIContent.none);
```

Třetí částí této metody je zobrazování label, tedy jeho vzhled a velikost. Pro label vytvoříme vlastní pozici Rect a snadno pak zobrazíme LabelField.

```
EditorGUI.LabelField(ImestoRect, "Město")
```




*Obrázek 38 - Property Drawer bez výšky*

Jak je vidět, nastal problém, kdy jsme vytvořili zobrazení, které zabírá více než jeden řádek a jak je vidět Unity samo neví, jak velké zobrazení jsme vytvořili. Proto pokud je zobrazovaný objekt jinak velký než jeden řádek, je potřeba vytvořit přetížení metody `GetPropertyHeight`.

```
public override float GetPropertyHeight(SerializedProperty property, GUIContent label)
{
    return base.GetPropertyHeight(property, label)+ EditorGUIUtility.singleLineHeight;
}
```

*Obrázek 39 - GetPropertyHeight*

Původní výšku poskytuje base verze této metody a velikost jednoho řádku lze zjistit pomoci `EditorGUIUtility.singleLineHeight`.

Takto upravený Drawer se již zobrazuje správně.


*Obrázek 40 - Kompletní Property Drawer*

#### 3.1.3.3 Inspektor

PropetyDrawery upravují vzhled „řádků“, neboli proměnných a tříd, které se zobrazí, avšak nezasahují přímo do objektů, které jsou zobrazovány v Inspektoru. Tím jsou na mysli komponenty typu MonoBehaviour nebo ScriptableObject [20].

Výhoda toho, že Inspektor zasahuje přímo do MonoBehaviour, nikoliv pouze do proměnných a dat, je v něm možnost ovlivňovat přímo chování objektů, či si přidat vlastnosti mimo data objektu.

Pro vytvoření tohoto okna je potřeba třídy dědící z Editor označené pomocí atributu CustomEditor [21], kterému předáme typ třídy, kterou zobrazuje. Poté přepsáním metody OnInspectorGUI určíme vzhled. Pokud chceme zachovat původní a pouze přidat vlastní obsah, lze využít původní base metody Editoru.

```
public class EditorExample: MonoBehaviour{}
[CustomEditor(typeof(EditorExample))]
Skript Unity | Počet odkazů: 0
public class EExampleEditor : Editor
{
    Počet odkazů: 0
    public override void OnInspectorGUI()
    {
        base.OnInspectorGUI();
    }
}
```

*Obrázek 41 - Editor Inspektor*

Samotné zpracování obsahu uvnitř Editoru je jednodušší než v PropertyDraweru. Jelikož se nejedná pouze o „proměnnou“, ale o komplexní zobrazení jedné komponenty, je Unity schopno na pozadí samo hlídat velikost a pozice objektů dle toho, jak je dáváme v pořadí do kódu. Není proto nutno vytvářet Rect objekty určující pozici a velikost.

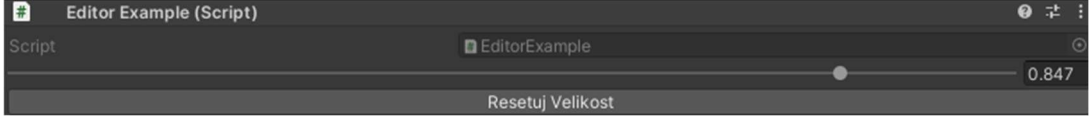
Místo EditorGUI se zde využívá EditorGUILayout, který si Rect spočte sám.

V této třídě je automaticky vytvořena proměnná target, která obsahuje odkaz na objekt, který tento inspektor edituje.

Tím můžeme ovlivnit i jiné věci než pouze data této komponenty. Příkladem může být vytvoření posuvníku, který není spojen s žádnou číselnou proměnnou, ale je naplněn současnou transform velikostí objektu a v daném rozmezí jí spočte a změní.

Také existuje možnost přidání vlastních tlačítek.

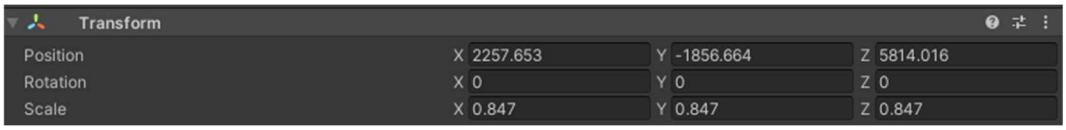
```
public override void OnInspectorGUI()
{
    base.OnInspectorGUI();
    EditorExample ee = target as EditorExample;
    var value = EditorGUILayout.Slider(ee.transform.localScale.x, .1f, 1f);
    if (GUILayout.Button("Resetuj Velikost"))
    {
        value = 1;
    }
    ee.transform.localScale = Vector3.one * value;
}
```




*Obrázek 42 - Tlačítko v inspektoru*

Tlačítko je uzavřeno v if podmínce, která je splněna ve chvíli, kdy je tlačítko zmáčknuté. Kvůli optimalizaci Unity hlídá změny nad všemi komponentami zobrazovanými v tomto Editoru a k volání OnInspectorGUI dochází pouze pokud na některých datech komponenty dojde ke změně. Díky tomu je možné mít v této metodě i výpočetně náročnější obsah. Tento obsah však nesmí změnit data komponenty na základě kterých by opět počítal. V takovém případě by došlo k zacyklení výpočtů a výsledku.

Předchozí ukázka skutečně při posunutí slideru, či kliknutí tlačítka změní Scale velikost na transformu objetku.




*Obrázek 43 - Transform*

Pokud MonoBehaviour v sobě obsahuje odkaz na jiné MonoBehavioura, je možné v Inspectoru využít i tato odkazovaná data. V příkladu je tímto způsobem přistupováno na transform. Pokud chceme mít jistotu, že GameObject, na kterém naše Behaviour je, tato data obsahuje, můžeme přidat atrubut RequireComponent nad definici naší třídy a tím zajistit, že opravdu existuje.

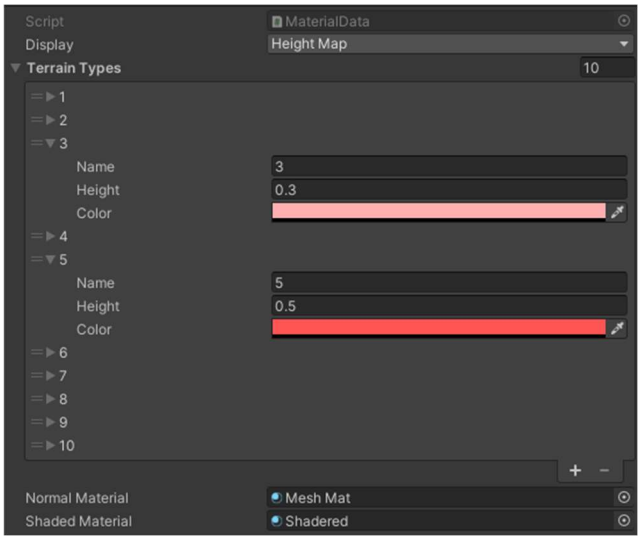
```
[RequireComponent(typeof(Transform))]
Skript Unity | Počet odkazů: 3
public class EditorExample: MonoBehaviour{}
```

*Obrázek 44 - RequireComponent*

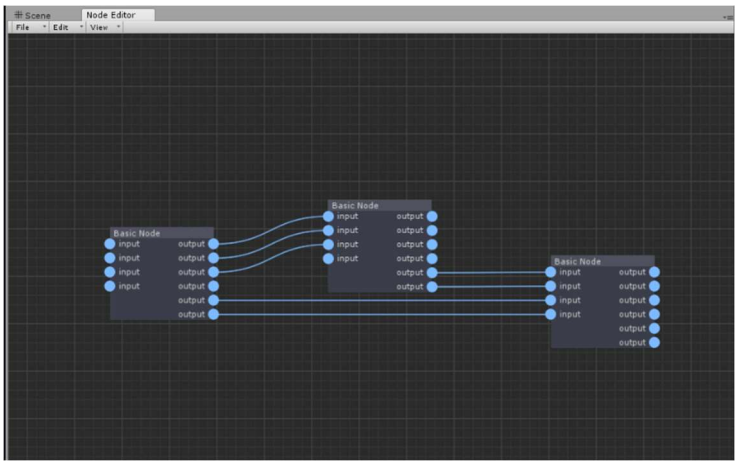
#### 3.1.3.4 Editor Widnow

Poslední možností vytvoření vlastního nástroje je editační okno. Příkladem takového okna jsou všechny moduly popsané v kapitole 3.4. Tato okna nejsou spjata s žádnou komponentou a mohou pracovat nezávisle na ostatních věcech v Enginu. Samozřejmě jsou vytvářeny na základě UnityEditor namespace, tedy nebudou nijak reprezentovány ve výsledném buildu.

Využití tohoto okna je velmi výhodné pro vytváření nástrojů, které zpracovávají větší množství obsahu nezávislému na jednotlivých scénách, či kontrolují velké množství stejných ScriptableObjektů. Také je zde možnost využití „komplexnějšího“ zpracování jednoho objektu, které není možné vytvořit v Inspektoru. Nejlepším příkladem může být využití Node based editačních oken poskytující lepší vizualizaci vztahů mezi daty.




*Obrázek 45 - ScriptableObject editor*



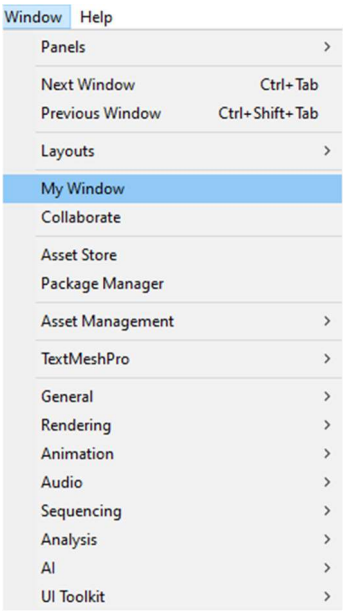

*Obrázek 46 - Node based editor*

Editační okno není závislé na žádném objektu. K jeho tvorbě stačí vytvoření vlastní třídy dědící z EditorWindow a vytvoření statické metody Init, která se zavolá při jeho zobrazení [22].

Zobrazení okna lze vyvolat kódově (z jiné Untiy komponenty), nebo pomocí atributu MenuItem, který poskytne tlačítko do Unity okna.

```
public class EditorWindowExample : EditorWindow
{

    [MenuItem("Window/My Window")]
    Počet odkazů: 0
    static void Init()
    {
        EditorWindowExample window = (EditorWindowExample)EditorWindow.GetWindow(typeof(EditorWindowExample));
        window.Show();
    }

    Zpráva Unity | Počet odkazů: 0
    void OnGUI()
    {
        GUILayout.Label("Hello world", EditorStyles.boldLabel);
    }
}
```

*Obrázek 47- EditorWindow kód*




*Obrázek 48 - MenuItem tlačitko*

Výsledkem je pak vlastní zobrazované okno, které lze, stejně jako každé jiné drag-and-drop, umístit kamkoliv v Unity programu.

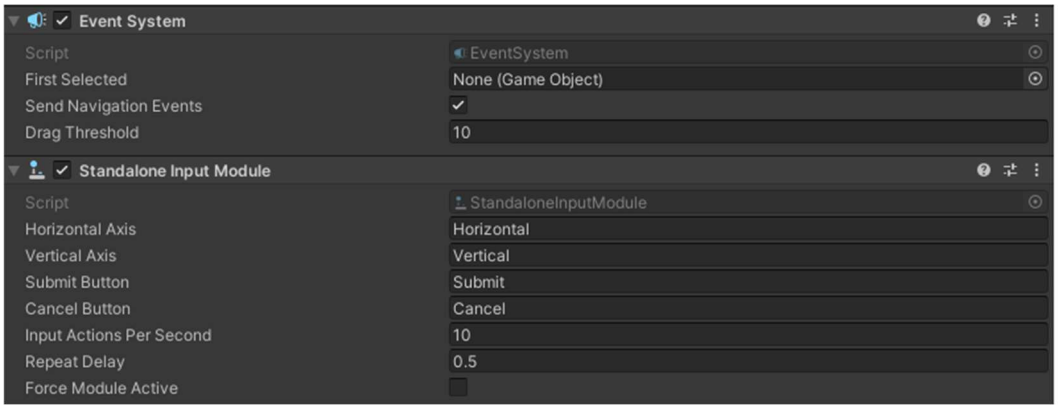
## 3.2 Správa událostí

Unity automaticky přebírá a poskytuje záznamy o situacích, které jsou způsobeny činností uživatele programu. Kódování je od těchto částí odděleno, jelikož tyto interakce jsou zpracovávány v různých chvílích cyklu program a do vlastních scriptů jsou poskytnuty pouze zaznamenané hodnoty zpracované interním kódem Unity [23].

### 3.2.1 Události uživatelského rozhraní

První skupinou událostí jsou ty, které pracují na 2D elementech UI rozhraní. Toto rozhraní je to nejjednodušší, jelikož všechny komponenty v UI vycházejí ze specifické verze GameObjectu, který obsahuje RectTransform místo Transform. Unity automaticky kontroluje interakce uživatele nad těmito komponentami.

Při vytvoření první komponenty UI ve scéně je také automaticky přidána komponenta zodpovídající za zpracování UI eventů.

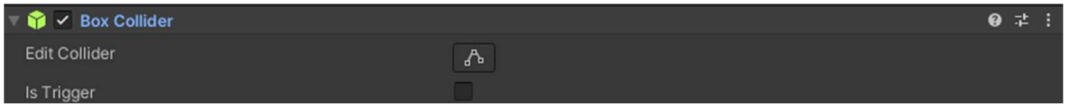



*Obrázek 49 - UI event systém*

Výsledky těchto událostí jsou díky tomu automaticky zpracovány v jednotlivých UI komponentách.

### 3.2.2 Události myši

Události nad 3D prostředím jsou více komplikované. V 2D zobrazení UI stačí pro pozici myši pouze její koordináty, ale ve třetím rozměru myší nepohneme. Unity to řeší tím, že vytvoří Raycast z pozice aktivní kamery na pozici myši, čímž vytvoří paprsek určující „pozici“ ve 3D prostoru. Objekty obsahující Collider komponentu dostávají od Unity informace ohledně interakce s myší a ty jsme poté schopni uvnitř MonoBehaviora využít [24].



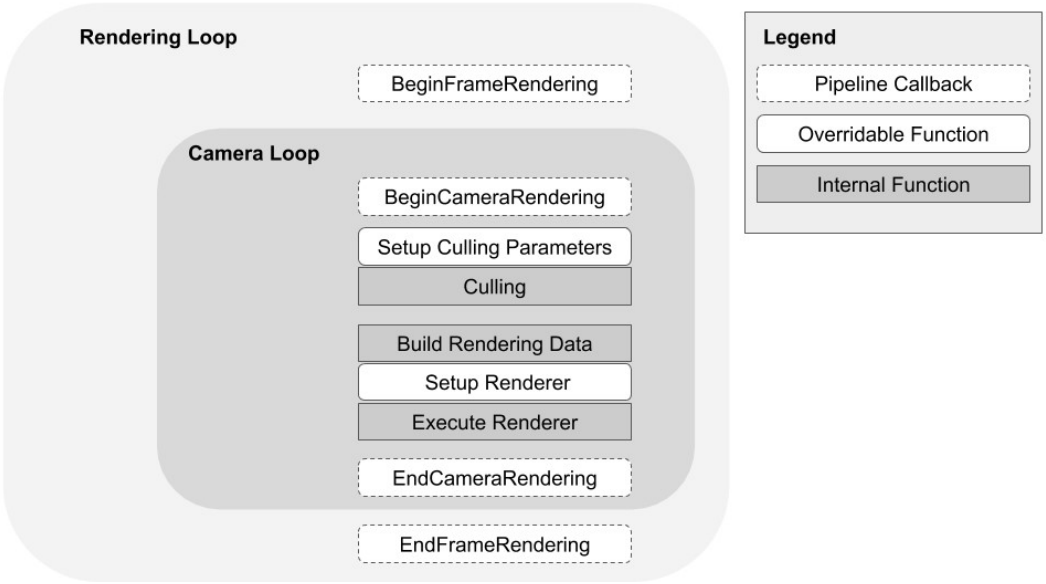

*Obrázek 50 - Collider*

K tomuto účelu obsahují scripty MonoBehaviour sérii metod jako je `OnMouseOver`. Ty jsou automaticky volány při splnění dané akce. Z tohoto důvodu nelze ověřovat interakce uvnitř Update metod, ale je nutné je nějak zaznamenat uvnitř OnMouse metod a na základě toho si určit data, která následující Update zpracuje.

## 3.3 3D Grafika

Pro zobrazení objektu na obrazovce je potřeba určit pravidla, dle kterých je daný objekt zobrazován. Tomuto postupu se říká Rendering Loop, jelikož jeden objekt může být „zobrazen“ několikrát, než dostane finální vzhled. Tyto kroky zobrazování mohou být

z důvodů překrytí objektů, využití určitých shaderů, nebo specifické potřeby, které si do Render Loop cíleně přidáme.

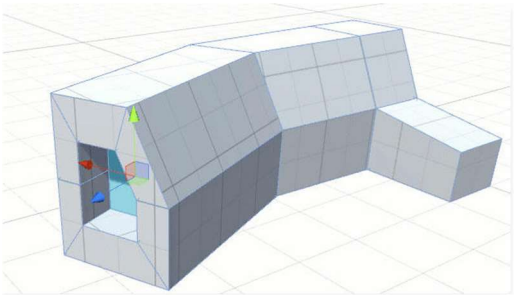



*Obrázek 51- Redner loop*
*(zdroj: https://docs.unity.cn/Packages/com.unity.render-pipelines.universal@10.4/manual/images/Graphics/Rendering_Flowchart.png)*

### 3.3.1 Mesh

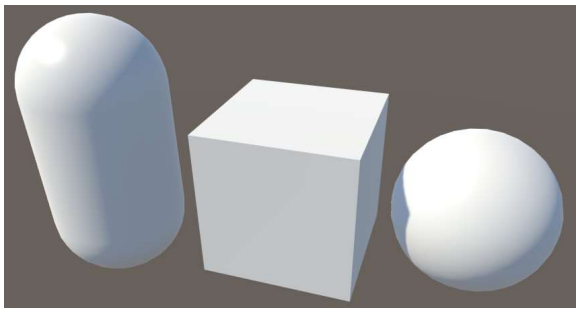
Pro zobrazení objektu je potřeba vytvoření dat, která reprezentují prostor zabíraný daným objektem. Tomuto objektu se říká model, nebo častěji mesh. Jeho tvorba může být pomocí 3D nástroje a do Unity jej pouze importujeme, nebo jej lze v Unity přímo vytvořit [25]. Vytvoření lze zajistit dvěma způsoby:

- pomocí interface s jednoduchými vlastnostmi, jako je velikost, rotace, extrude a inset stěn či hran,

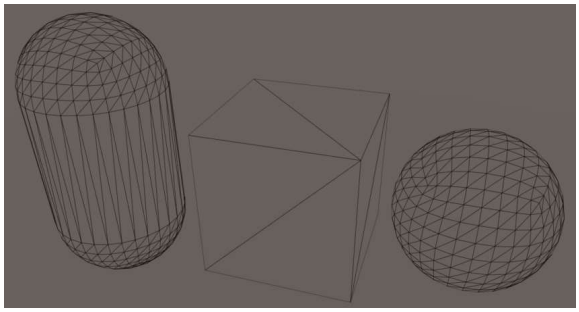


*Obrázek 52 - Ruční tvorba mesh*

Jakýkoliv mesh je reprezentován pomocí série trojúhelníků neboli polygonů. Každý polygon má rovné hrany a spojuje tři vrcholy v daném pořadí a tím vytváří mezi nimi stěnu. Kvůli tomu nelze přesně reprezentovat zaoblené objekty, jejich tvar lze pouze napodobit.



*Obrázek 53 - Mesh objekty*

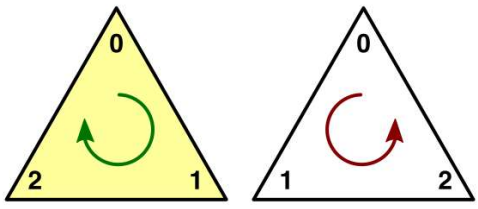


*Obrázek 54 - Wireframe Mesh Objektů*

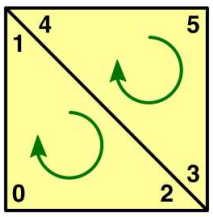
Datově je mesh reprezentován dvěma poli číslic:

- jednotlivé vrcholy mesh v prostoru,
- indexy z prvního pole, kde každé 3 po sobě jdoucí číslice znamenají propojení mezi nimi a tím vytvoření jednoho polygonu.

Pořadí číslic ve skupině reprezentující polygon je důležité. Pokud není určeno jinak, je trojúhelník zobrazován pouze z jedné strany určenou pořadím vrcholů ve směru hodinových ručiček.



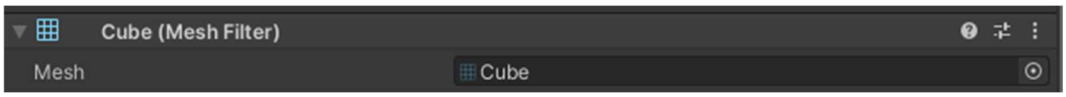

*Obrázek 55 - Vrcholy Polygonu*




*Obrázek 56 - Vrcholy Quad*

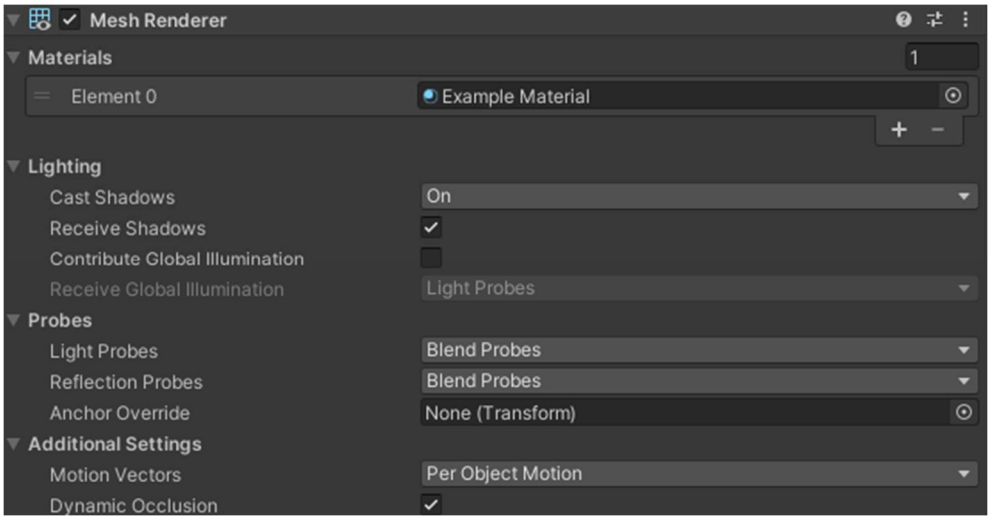
Každý objekt fyzicky zobrazovaný uvnitř scény je v Unity reprezentován za pomoci GameObjektu, který musí obsahovat dva MonoBehaviour scripty.

MeshFilter je script odkazující na fyzický mesh objekt, který je uložen někde v hierarchii projektu. Tím dochází k ušetření velikosti dat, jelikož všechny objekty, které jsou založeny z jednoho meshe, sdílejí tento odkaz.




*Obrázek 57 - Mesh Filter*

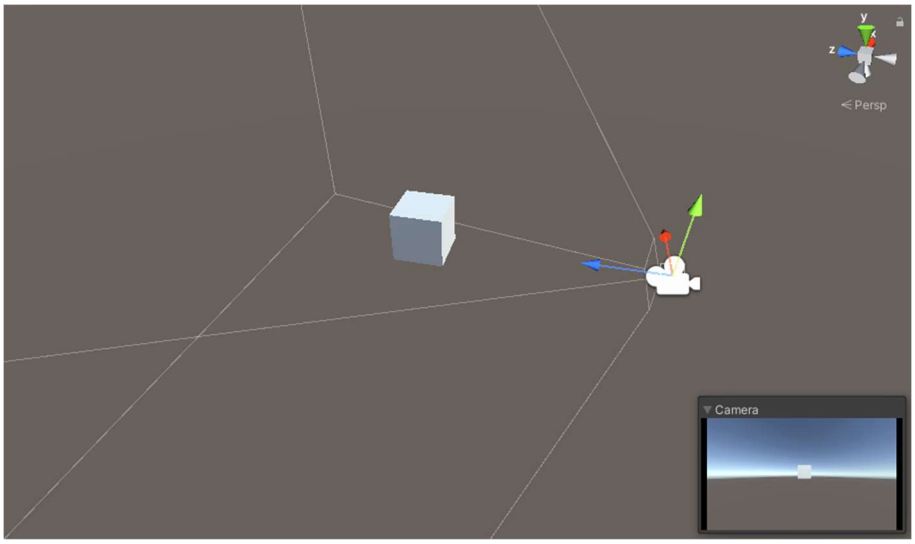
Druhým scriptem je MeshRenderer, který určuje způsob, jakým se má daný objekt zobrazovat. Základem je materiál určující vzhled. Zbylé součásti Rendereru nastavují interakci s prostředím a světlem.




*Obrázek 58 - Mesh Renderer*

### 3.3.2 Kamera

Pro zobrazování je potřebné určit pohled, ze kterého se budou objekty sledovat. K tomu slouží script Camera, který pomocí svého transformu (pozice, rotace a velikost) určuje pozici pro specializované parametry kuželu určujícímu zobrazovaný prostor. Také poskytuje zdrojový bod pro Raycasting paprsky, které se směrem k těmto objektům budou vysílat.




*Obrázek 59 - Kužel kamery*

### 3.3.3 Culling

Culling jsou pravidla určující zobrazované objekty, ale i pravidla, jaké objekty v okolí je mohou ovlivňovat. Obecně se jedná o jakýkoliv proces odebírající objekty z následujícího vykreslování, které nebude potřeba zobrazit, protože nijak neovlivní výsledek viděný na kameře. Nejedná se však o pouhé „ignoruj vše co není v zobrazení“. Příkladem mohou být světla, která jsou mimo kameru, ale ovlivňují nasvícení zobrazovaného objektu, a také jakýkoliv předmět mimo kameru tato světla blokující [26].

Cílem tohoto procesu je snaha o maximální snížení množství vykreslovaných objektů tak, aby se ušetřil výkon. Standartním postupem je:

1. odstranění objektů mimo zobrazení,
2. vrácení vyřazených objektů ovlivňujících vzhled zobrazovaných objektů nepřímo,
3. odstraní nepotřebných části objektů, tedy těch, které nejsou vidět ať už z důvodu orientace, nebo zakrytí jiným objektem.

Frustrum culling se nazývá technika, během které se odstraní ze seznamu objektů k vykreslení všechny, které nejsou v pohledovém kuželu kamery.

Back-face culling se stará o odstranění odvrácených stran objektů.

Z-test nakonec odstraní překrývající se části objektů. K tomu si zjistí pro každý polygonem zakrytý pixel na obrazovce vzdálenost mezi kamerou a tímto polygonem. Polygony, jejichž body jsou zakryty, se odstraní. Zakrytí je určeno tím, že existují bližší polygony ke kameře na stejném pixelu.

### 3.3.4 Osvětlení

Pro určení zobrazovaného světla existují dvě posloupnosti vykreslování.
Forward rendering je jednoduchá technika, kdy je světlo načteno jako součást shaderu pro zobrazovaný objekt.
Deferred rendering proti tomu využívá výpočet světla až po z-testu, a to ve chvíli, kdy je veškerá geometrie připravena k vykreslení. Díky tomu může ovlivnit objekt i více zdroji světla současně.

### 3.3.5 Vykreslovací data

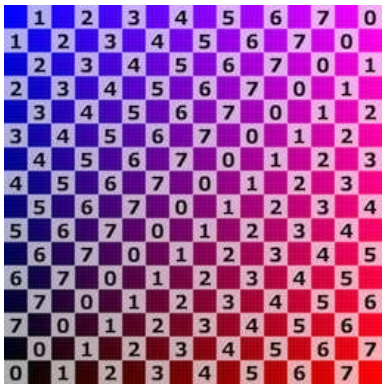
Když je Cullingem určeno, jaká data se mají zobrazovat, je potřeba určit způsob jejich zobrazení. To určuje jejich mesh geometrie, materiál a shadery. Díky tomu, že tato data jsou velmi často neměnná, bývají grafické kartě zaslána jen v případě, že některou z informací nemá, nebo byla změněna. V opačném případě jsou v paměti grafické karty ponechána pro více snímků, pouze se mění pozice zobrazení.
V procesoru nemusí být uchovávána celá data, může udržovat jen odkazy na objekty v grafické kartě. Než se odešlou informace specifické pro daný snímek, procesor seřadí objekty a určí tak jejich pořadí při vykreslování. V určování pořadí je důležitá vzdálenost od kamery a zda je či není objekt průhledný. Neprůhledné objekty jsou řazeny od nejbližších po nejvzdálenější, průhledné objekty od nejvzdálenějších po nejbližší. Na grafickou kartu se pak pošle zbytek informací a začne se vykreslovat.

### 3.3.6 Následné zpracování (Post-processing)

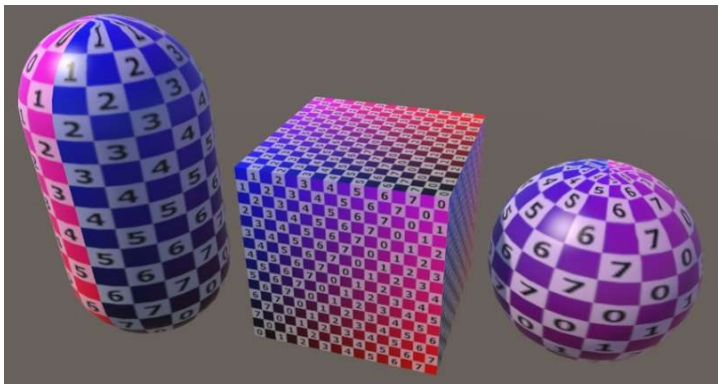
Posledním, ale nepovinným krokem, je Post-processing. Pokud je aktivní, tak se předchozí kroky vytvářejí pouze do grafické paměti a na obrazovku jsou umístěny až poté, co se použijí shadery pro aplikování Post-processingu. Tyto filtry mohou potřebovat i několik průchodů. Protože se již jedná o poslední krok vykreslování, existuje zde možnost poskytnout filtrům přístup k celému vykreslenému obrazu [27].

### 3.3.7 Materiál

Materiál určuje způsob, jakým je daný objekt vykreslen. K tomu je využito mapování UV souřadnic uložených v mesh a určujících „zobrazení“ 2D textur obsažených v Materiálu na 3D objekt meshe [28].

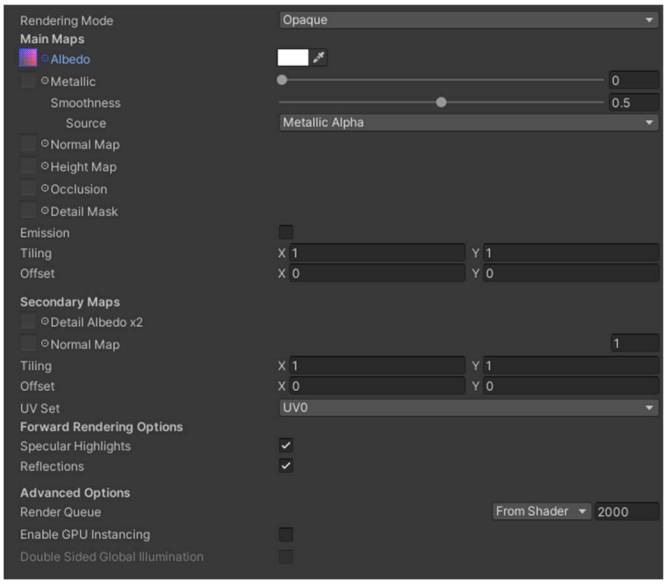


*Obrázek 60 - UV Textura*



*Obrázek 61 - UV Mesh*

Materiál obsahuje sérii textur, které dohromady napodobují interakci světla s objektem [29].



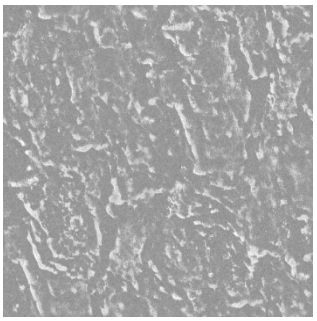

*Obrázek 62 - Material*

1) **Albedo** je RGB mapa, která obsahuje dva typy dat:
   - reflexní barvy pro dielektrické materiály,
   - hodnotu odrazivosti pro metalické materiály.

   Jeden materiál a tedy i jedna textura může současně obsahovat oba typy. Příkladem může být poškrábaný kov, kde kov je metalický, ale jeho vady jsou dielektrické.

*Obrázek 63 - Textura | Albedo*

2) **Metalic** je grayscale mapa určující rozlišení, které části jsou dielektrické a které metalické. Mapa poskytuje pouze rozlišení mezi dvěma styly a většinu času by měla být pouze ve dvou barvách černá/bílá, protože nic nemůže být současně dielektrické i metalické. Hodnoty mimo 0 a 255, tedy odstíny šedé, se v praxi ale využívají pro reprezentaci zakrytí metalického materiálu pomoci dielektrického. Příkladem může být rez na kovu.

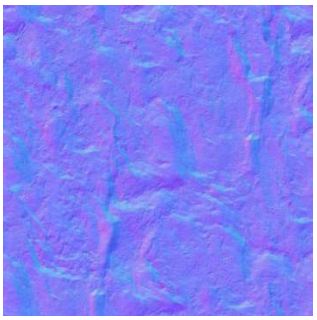


*Obrázek 64 - Textura | Metalic*

3) **Normal map** je textura určující nerovnosti na rovném povrchu. Je vhodná k simulaci detailů na povrchu objektu, pro který by byl potřeba extrémně velký počet polygonů, aby byly přesně reprezentovány meshem. Při artistické tvorbě jsou často vytvářeny přemalováním velmi detailní verze modelu na low polygon model, kde tato mapa poté nahrazuje ztracené detaily.

   Každý pixel na této mapě reprezentuje odchýlení od rovného povrchu definovaného polygony. Prostorové souřadnice odchylek v osách XYZ jsou mapovány na jednotlivé RGB kanály textury:

- R (červená) 0 až 255 reprezentuje osu X: -1 až 1
- G (zelená) 0 až 255  reprezentuje osu Y: -1 až 1
- B (modrá) 128 až 255  reprezentuje osu Z: 0 až 1

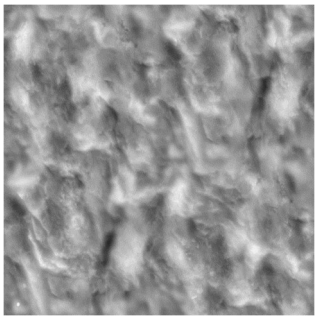


*Obrázek 65 - Textura | Normal*

Polygon zobrazovaný směrem od kamery není kamerou viděn, hodnoty < 0 by způsobily chyby v zobrazování. Hodnoty osy Z a tedy i modrého kanálu jsou díky tomu kladné. To způsobuje, že je textura vždy v nějakém odstínu modré.

Změny provedené pomocí normal mapy jsou pouze vizuální a spočtené jako interakce světla s objektem.

4) **Height mapa** je podobná normálové mapě, ale její princip je složitější a tím i pro finální renderování náročnější. Na rozdíl od „vizuálních“ změn u normálové mapy dochází u Height map k opravdovým pohybům nad souřadnicemi meshe. Tím přidává skutečnou hloubku do objektu.



*Obrázek 66 - Textura | Height*

5) **Ambient Occlusion** je jednokanálová grayscale mapa reprezentující záznam stínů, která poskytuje simulaci globálního světla. Využívá se pro „ztmavení“, nebo omezení působení světla v oblastech materiálu. Tím simuluje spoje objektů, rohy či hlubší části objektu, kam se světlo hůře dostane.

*Obrázek 67 - Textura | Ambielt Occlusion*

6) **Detail Mask** je textura vytvářející masku s určitou průhledností pro zobrazení sekundárních textur se specifickými detaily.
    - Secondary Maps – Sekundární mapy jsou využity mapou detailů.

### 3.3.8 Shader

Po určení dat objektů, které se mají zobrazovat a specifikování textur ovlivňujících jejich vzhled, je ještě potřeba určit způsob, jakým k tomuto zobrazení dojde. K tomuto účelu slouží shader. Ten určuje deterministická pravidla pro vytvoření skutečné reprezentace daného pixelu na obrazovce. Do shaderu se předají informace o daném pixelu a ten určí jeho barvu. Je to kompletně deterministická metoda, která je nezávislá na okolí. Lze tedy paralelně vypočítat zobrazování každého pixelu, což je důvod proč GPU dosahují takového výkonu pro generování grafiky. Jeden pixel může být zpracováván i více shadery, kde každý vygeneruje svoji verzi výsledného obrázku. Technicky vzato jednotlivé shadery negenerují celý obraz [30].

Existuje několik variant shaderů s různými účely:

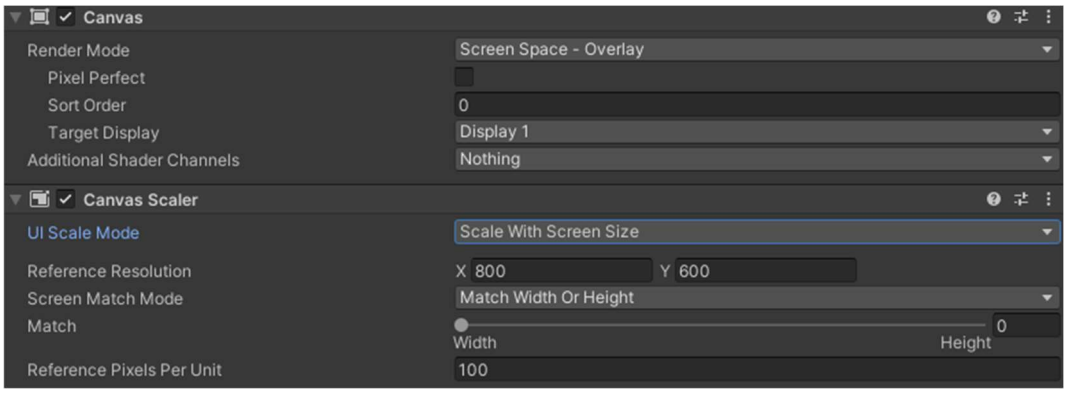
- Pixel Shader, kde výsledkem je pixel a jeho pozice na obrazovce.
- Fragment Shader produkuje barvu jednoho pixelu, který je spočten na základě odpovídajících materiálů a světel ve scéně.
- Vertex Shader vypočítává pozicování pixelů proti jeho „datové“ pozici. Je využit například u Height map.
- Compute Shader nevytváří nic vizuálního, ale poskytuje možnost provést paralelní výpočty probíhající na GPU.

Kombinací těchto kroků je vytvořen finální produkt, který může 3D nástroj zobrazit na obrazovku uživatele.

## 3.4 Uživatelské rozhraní

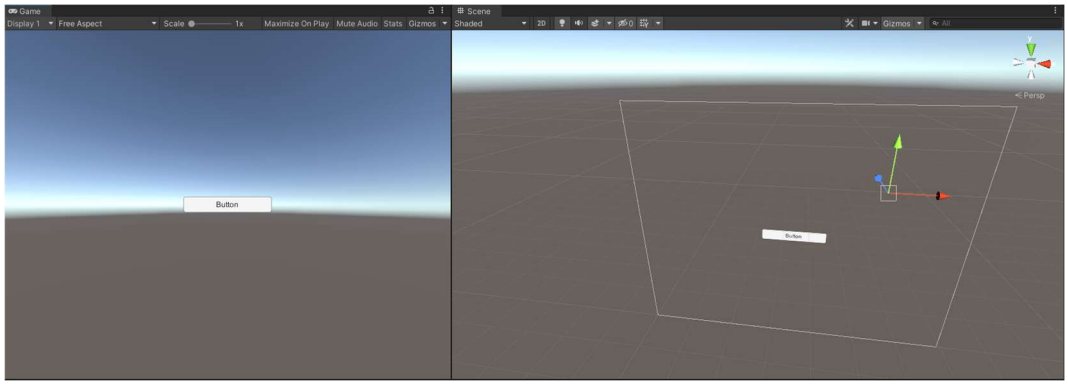
Uživatelské rozhraní (UI) může být v Unity reprezentováno dvěma způsoby. V prvním jsou objekty zobrazované jako překrytí na obrazovce, ve druhém jsou 2D objekty umístěné ve 3D prostoru. UI v prostoru je řešeno pomocí standartní Render Loop, není potřeba je pro UI více popisovat [7].

Objekty zobrazované na obrazovce jsou určeny pomocí objektu Canvas, který funguje jako plátno určující velikost vykreslovaného okna. Díky tomu je jejich zobrazení naprosto nezávislé na kameře, která řeší prostor.



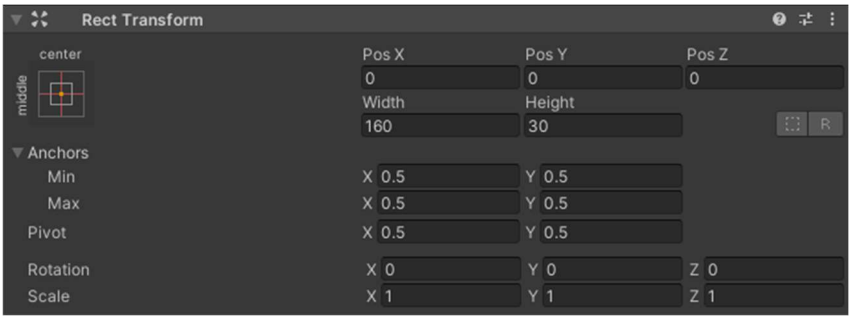

*Obrázek 68 - Canvas*

Tyto objekty jsou sice v prostoru zobrazeny, ale kamera je ignoruje a na obrazovku jsou umístěny jako by jejich Canvas byl přesně zobrazovanou obrazovkou.




*Obrázek 69 - Canvas v prostoru*

Pro určování rozměrů a pozice jednotlivých UI komponent existuje speciální varianta Transformu, tedy RectTransform. Funguje na principu subsetu kaskádových stylů, kterým je možné nastavit vzhled dané komponenty, tj. velikost, pozice, ukotvení, ale i rotace v 3D prostoru.

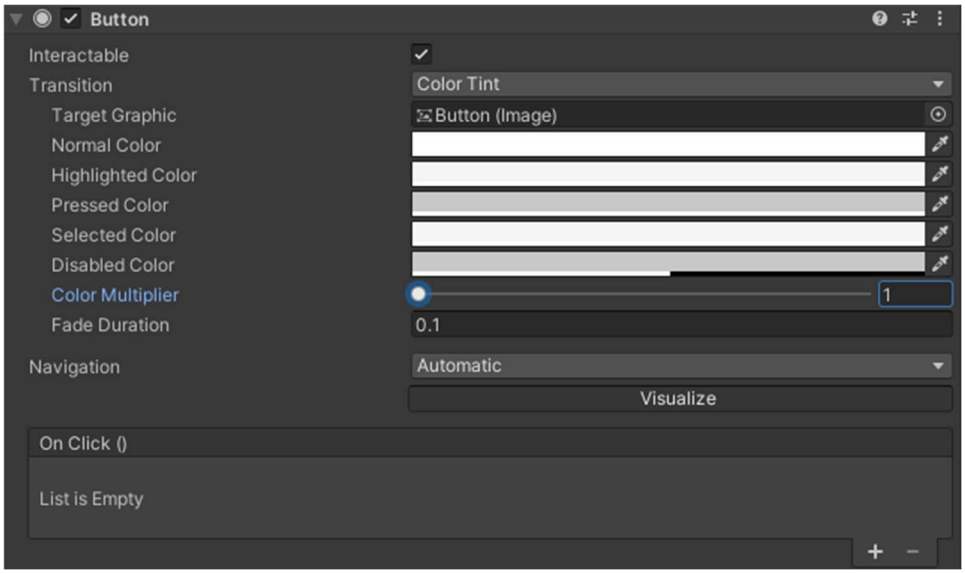



*Obrázek 70 - Rect Transform*

Unity obsahuje skupinu specifických komponent, jejichž složením lze vytvořit libovolné UI. Chování těchto objektů je určeno jejich vlastními MonoBehavioury, k nimž lze přistupovat stejně jako k ostatním, tedy pro scripty není rozdíl, zda jde od UI či 3D objekt. Některé komponenty jsou předem vytvořeny jako kompozice několika jiných, například všechny věci obsahující text v sobě obsahují jako potomka textovou komponentu. Tato textová součástka je zde ve dvou verzích:

- Text – rastrovaný text,
- TextMeshPro – vektorový text.

Kvůli tomu jsou zde současně dvě verze i od věcí, které mají textového potomka. Scripty reprezentující tyto UI obsahují napojení na jednotlivé události, které mohou spustit.



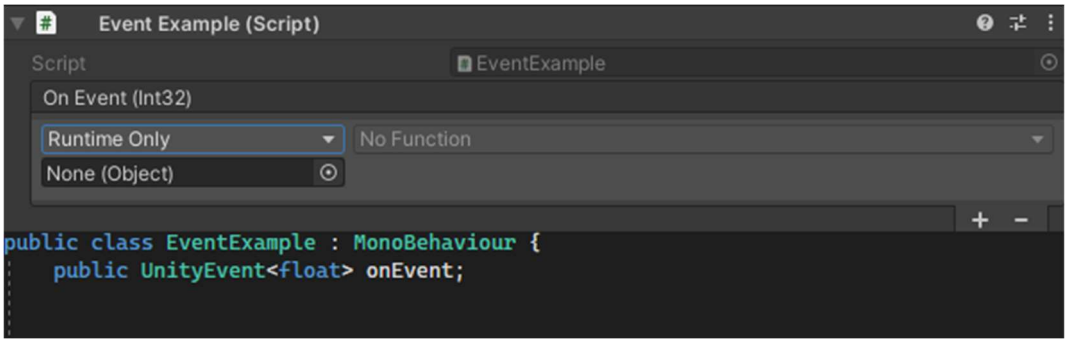

*Obrázek 71 - Button*

Pro očekávané události je připravený Inspektor na změny, například barvy u různých stavů tlačítka. Existují ale i možnosti specifikách změn. Pro ty je zobrazen UnityEvent

spuštěný při dané události. Do tohoto objektu lze přiřadit vlastní metodu včetně parametrů.

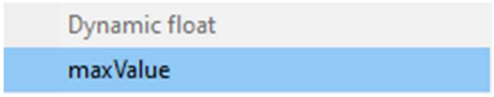

*Obrázek 72 - UnityEvent*

Lze také vytvořit vlastí události spustitelné pomocí `event?.Invoke()`, čímž se provedou všechny napojené metody. Napojení lze provést jak v Inspektoru, tak i kódově pomocí `event+=metoda`.



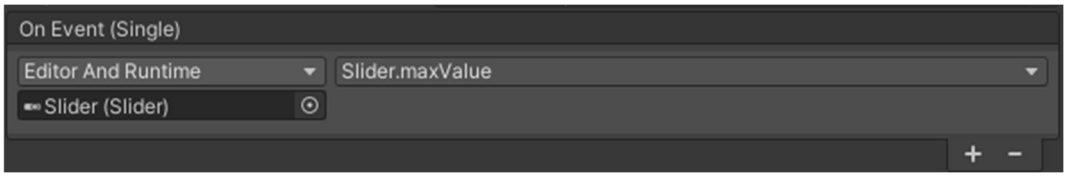

*Obrázek 73 - Vlastní UntiyEvent*

Je vidět, že u eventu jsme určili typ, který může poskytovat hodnotu do metody přímo z místa Invoke. V předchozím příkladu je využit float: `onEvent?.Invoke(123456)` poskytující hodnotu do napojených metod a díky tomu lze poskytnout dynamickou hodnotu, která je automaticky předána.




*Obrázek 74 - Dynamická hodnota*

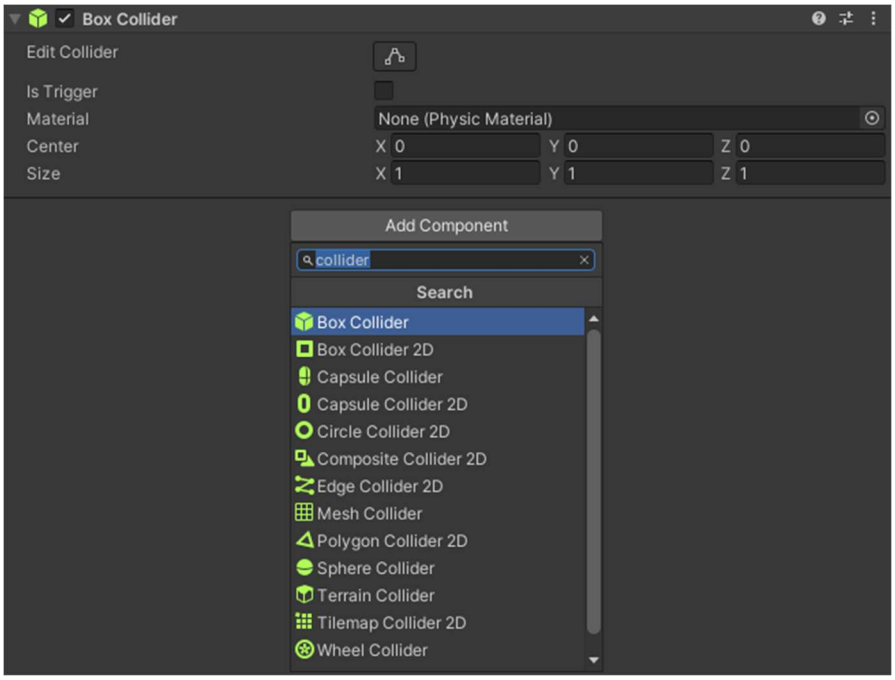

*Obrázek 75 - Event staticky*




*Obrázek 76 - Event dynamicky*

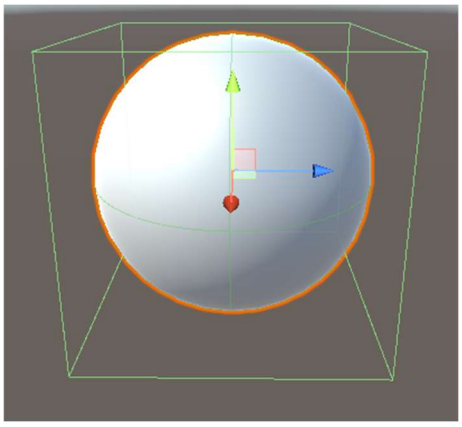
## 3.5 Fyzika

Pro interakci objektů mezi sebou obsahuje Unity sérii MonoBehaviour, které na pozadí kontrolují svoje pozice navzájem a současně proti pozici myši. Tyto komponenty se nazývají Collider a určují fyzickou hranici, která je pro objekt hlídána [31].




*Obrázek 77 - Existující Collidery*

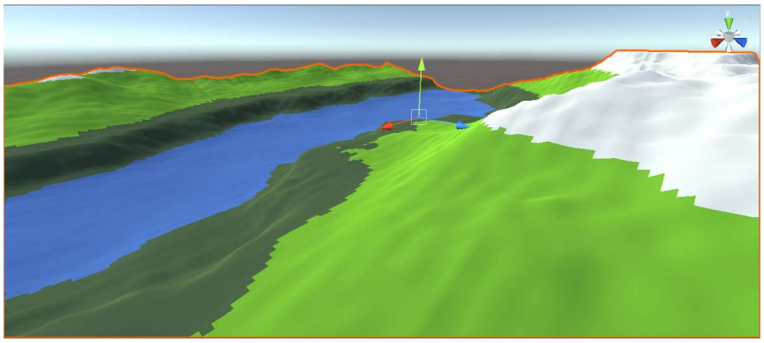
Jsou vytvořeny ve verzích 2D a 3D. Obě varianty fungují stejným způsobem, jediným rozdílem je, že pro 2D není hlídána depth (z) osa, a tím jsou méně náročné na zpracování. Kolizní hranice objektu nemusí vůbec souviset se zobrazovaným objektem.



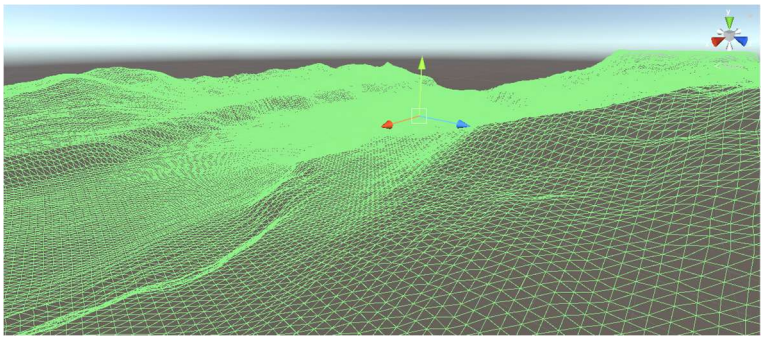
*Obrázek 78 - Koule s Collider Boxem*

Tím se dá poměrně zjednodušit náročnost kontrolovaných hranic, jelikož pro velmi komplexní objekt lze hlídat pouze jeho „přibližný“ tvar nasimulovaný pomocí primitivních Colliderů. Nicméně pokud je potřeba dodržet interakci velmi přesnou a

podobnou skutečnému tvaru objektu, existuje varianta MeshCollider, která přijímá mesh objekt jako svůj zdroj a jeho polygony využije pro vytvoření kolizních hran.

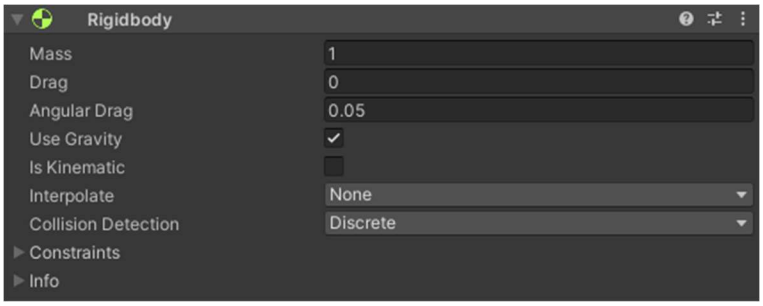


*Obrázek 79 - Kolize Mesh*



*Obrázek 80 - Kolize Collider*

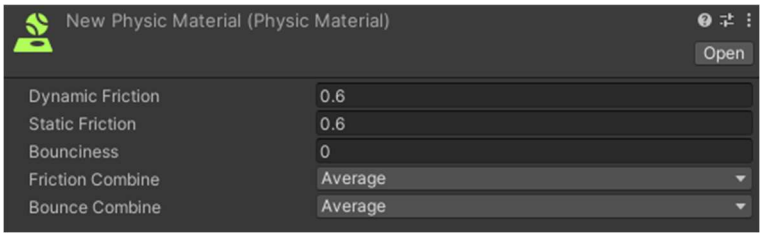
Od objektů s Collidery získáme pouze informace o tom, jaké objekty se navzájem dotýkají, avšak samotnou kolizi nijak neřeší. Jsme schopni posunout jeden Collider do druhého a nic se nestane. Aby začal objekt skutečně reagovat na tyto kolize je potřeba všem objektům, které aktivně mění situaci kolize (tedy ty, které nejsou pevně statické) přidat objekt typu Rigidbody.




*Obrázek 81 - Rigidbody*

Tento script zpracovává způsob, jak na sebe jednotlivé objekty reagují. Jejich hmotnost, rychlost zpomalení a zda využívají gravitaci. Pokud je ve scéně objekt, který se sice časem mění, ale jeho změny jsou tvořeny mimo tento fyzikální systém, existuje zde možnost `IsKinematic`, která informuje ostatní ovlivněné objekty. Jeho samotného

ostatní ale změnit nemohou. V našich příkladech je toto přesně případ u terénu, se kterým budou ostatní věci interagovat, ale opravdu by s ním neměly být schopny pohnout.
Různé objekty mají různé způsoby interakce chování. V realitě je tato interakce určena tvarem a složením daného objektu, ale jelikož naše Collidery pouze simulují realitu, je potřeba jim nějak říci jaké by jejich skutečné chování bylo. Například SphereCollider je dokonalá koule, což ve skutečnosti není možné. Povrch koule má určité nedokonalosti, které změní jeho tření s okolím a proto existuje Physic Material.

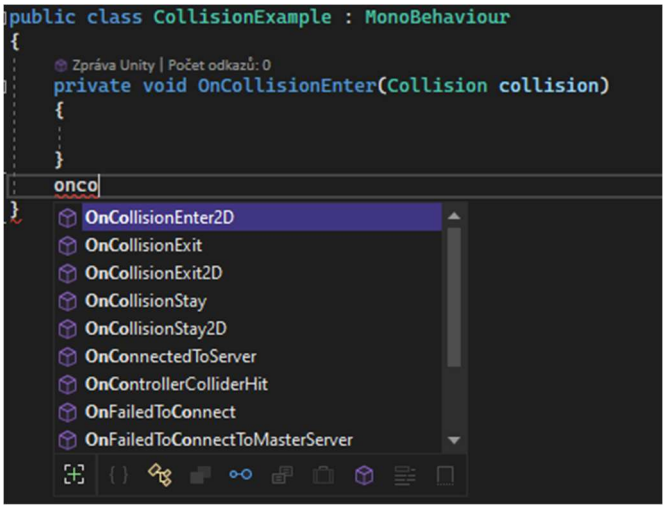



*Obrázek 82 - Physic Material*

Ten se vytváří jako objekt v Hierarchii a lze jej přiřadit jakémukoliv Collideru. V tomto materiálu lze určit jakým způsobem se objekt odráží a jakým ztrácí rychlost. Frikce je hlídána dvěma způsoby:

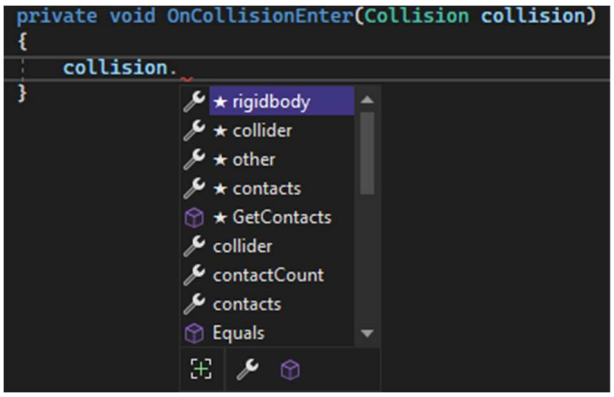
- Dynamic Friction je využita, když je objekt v pohybu. Jedná se o hodnotu 0-1, kde 0 je velmi hladký objekt, a 1 je drsný objekt, který se rychle zastaví.
- Static Friction se využívá, pokud je objekt nehybný. Hodnota je v rozmezí 0-1 a určuje náročnost na sílu potřebnou k tomu, aby se objekt dal do pohybu.

Aby se lépe aproximoval skutečný tvar objektu lze jeden objekt reprezentovat více Collidery. Interakce mezi kolizemi je řešena automaticky. Není potřeba ve vlastních scriptech složitě propojit všechny související Collidery, aby systém věděl, že jsou z jednoho objektu a tedy by neměly spolu interagovat. Také na nich není potřeba hlídat změny. Ty jsou automaticky poskytnuty všem souvisejícím MonoBehaviour do jejich metod OnCollision*. Metody jsou spouštěny situacemi z jakéhokoliv Collideru na stejném objektu.




*Obrázek 83 - OnCollision metody*

Z objektu kolize, který všechny tyto metody dostávají, získáme informace o obou Colliderech, které tuto interakci vyvolaly. Poskytuje i odkaz na rigidbody script a také lze zjistit velikost kolizních hran.




*Obrázek 84 - Collision objek*

# 4 Implementace

Pro vývoj praktické části práce byla vybrána nejnovější LTS verze Unity 2020.3. LTS je verze s dlouhodobou podporou. Vývoj byl řešen pomocí iterativního přístupu, kdy se vývoj rozdělil na postupně složitější mechanismy a tyto jednotlivé kroky byly postupně vytvořeny. Systém se sestavuje z několika MonoBehaviour, kde každé z nich má určen přesný úkol na řešení. Díky tomu lze snadno testovat tyto kroky a poskytuje to také možnost si zobrazit výsledky jednotlivých kroků, nejen celkový výsledek.

## 4.1 Generování mesh

Prvním krokem bylo generování základního meshe. Pro vytvoření meshe je potřeba vytvoření 3 datových prvků:

- Vrcholy
- Trojúhelníky
- UV

Jelikož pro terén postačí pouhá plocha rozdělena na čtverce, může náš základní mesh vzniknout určením počtu rozdělovacích bodů. Stejný počet je i pro UV. Ty určují způsob mapování textur. Posledním prvkem jsou trojúhelníky. Jak bylo řečeno, ty fungují jako posloupnost 3 hodnot pro jeden trojúhelník. Proto je potřeba vytvoření $(Výška - 1) \times (Šířka - 1) \times 6$ hodnot.

```
public class TerrainMeshData
{
    public Vector3[] Verticies;
    public int[] Triangles;
    public Vector2[] UVs;

    Počet odkazů: 1
    public TerrainMeshData(int terrainWidth, int terrainHeight)
    {
        Verticies = new Vector3[(terrainWidth * terrainHeight)];
        UVs = new Vector2[(terrainWidth * terrainHeight)];
        Triangles = new int[(terrainWidth - 1) * (terrainHeight - 1) * 6];
    }
```

*Obrázek 85 - Tvorba Meshe*

Vrcholy jsou posouvány o jednu pozici po osách X,Y. Tím se vytvoří pravidelná mřížka bodů.

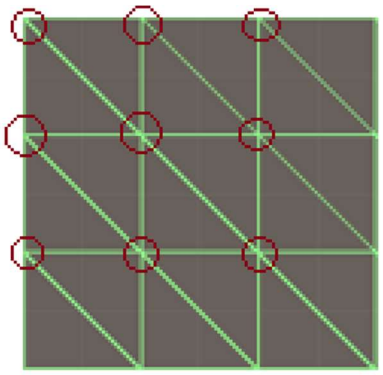
```
for (int y = 0; y < terrainHeight; y++)
    for (int x = 0; x < terrainWidth; x++)
    {
        terrainMesh.Verticies[vertexIndex] = new Vector3(topLeftX + x, 1, topLeftY - y);
        vertexIndex++;
    }
```

*Obrázek 86 - Pozice vrcholů*

Pro určení vrcholů tvořící uskupení lze využít postup, kdy pro každý vrchol, za nímž oběma směry (doprava a dolů) existuje alespoň jeden další vrchol, máme jistotu, že tvoří 2 trojúhelníky. Proto projdeme přesně tyto vrcholy a za každý z nich vytvoříme čtverec. Tyto pozice jsou postupně v listu, to znamená, že první vrchol druhého řádku je na pozici $0 + \text{šířka}$. Kvůli zachování směru po hodinových ručičkách, tak aby byly všechny vrcholy orientovány stejným směrem, jsou pozice těchto dvou trojúhelníků:

$$T1 = pozice + \text{šířka}, pozice, pozice + 1$$

$$T2 = pozice, pozice + \text{šířka} + 1, pozice, pozice + \text{šířka}$$



*Obrázek 87 - Procházené body meshe*

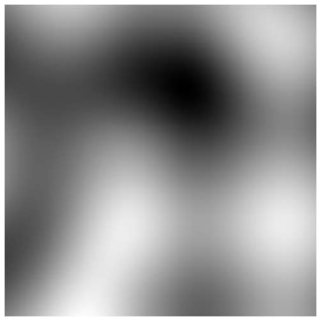
## 4.2 Generování terénu

Na základě předchozího kroku máme vytvořenou plochu. Pro tu je však potřeba určit její posun po ose Z. Na určení tohoto postupu využijeme generování Perlinova šumu.
Mapa je pak vytvořena jako pole float hodnot na X,Y pozicích.

```
public static float[,] GenerateNoiseMap(int mapWidth, int mapHeight, float scale)
{
    float[,] noiseMap = new float[mapWidth, mapHeight];
    for (int y = 0; y < mapHeight; y++)
        for (int x = 0; x < mapWidth; x++)
        {
            float perlinValue =(float) Perlin.noise(x, y, scale);
            noiseMap[x, y] = perlinValue;

        }
    return noiseMap;
}
```

*Obrázek 88 - Kód generování noise*

Samotný šum je vytvářen na základě souřadnic a výškové hodnoty, která je použita jako násobitel výsledku. Proces prováděný v této metodě byl popsán v kapitole 2.1.



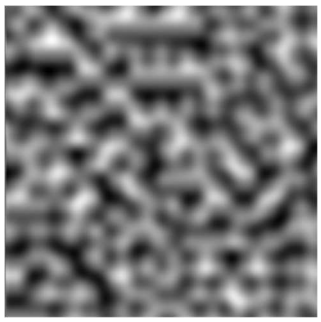
*Obrázek 89 - Perlin Noise*

Šumu poté přidáme detaily pomoci principu oktáv, kde jsou vytvořeny proměnné:

- detailPersistance určující, jak moc menší oktávy ovlivní jejich amplitudu,
- detailLacunarity určující, jaká je frekvence těchto detailů.

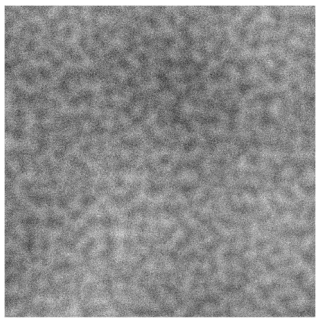
```
for (int i = 0; i < octaves; i++)
{
    float perlinValue = (float)Perlin.noise(x, y, scale*frequency);
    noiseHeight += perlinValue*amplitude;

    amplitude *= detailPersistance;
    frequency *= detailLacunarity;
}
noiseMap[x, y] = noiseHeight;
```

*Obrázek 90 - Kód oktáv*

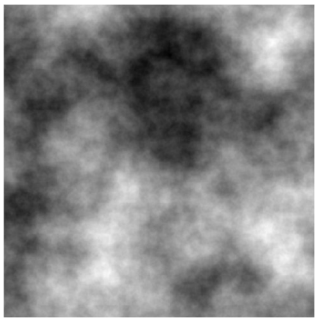


*Obrázek 91 - Noise o 7 oktávách*

Při různých hodnotách proměnných vznikají velmi odlišné vzhledy šumu.



*Obrázek 92 - Perlin Noise s Persistance 1 a Lacunarity 15*



*Obrázek 93 - Perlin Noise s Persistance 0,45 a Lacunarity 2,5*

Jelikož velké extrémy hodnot vytvářejí mapy, které se blíží náhodnému šumu, je vhodné zařídit omezení těchto hodnot. K tomu lze využít Range atribut omezujícím jejich rozsah.

```
[Range(1f, 10f)]
public int octaves;
[Range(0f, 1f)]
public float detailPersistance;
[Range(1f, 10f)]
public float detailLacunarity;
```

*Obrázek 94 - Range omezení šumu*

Také je vhodné vytvořit omezení pro rozměry šumu. Ty by se nikdy neměly dostat do záporu, tím by program mohl vyhodit výjimku.

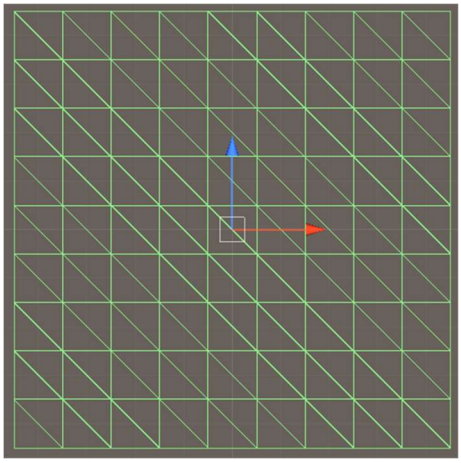
MonoBehavioura a ScriptableObjekty poskytují metodu OnValidate, která je volána kdykoliv dojde ke změně dat v tomto objektu.

Perlin noise, který doposud vytváříme, je tvořen deterministicky na základě předvytovřené tabulky zvané looktable. Tím je pro stejné nastavení vždy tvořen stejný výsledek. To je dobrá věc, ale omezuje nás v množství variant generovaného terénu. Tento šum je však teoreticky nekonečný a díky tomu lze vytvořit větší variaci pomocí posunů. Tento posun je tvořen na základě generátoru náhodného čísla, kterému je poskytnuta seed hodnota, aby šlo zajistit opakované generování stejného výsledku.

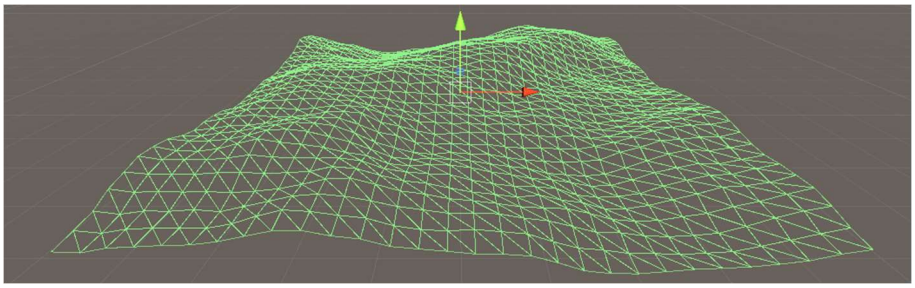
```
System.Random rand = new System.Random(seed);
Vector2[] octaveOffsets = new Vector2[octaves];
for (int i = 0; i < octaves; i++)
{
    float offsetX= rand.Next();
    float offsetY = rand.Next();
    octaveOffsets[i] = new Vector2(offsetX, offsetY);
}
```

*Obrázek 95 - Random Seed*

Vytvořené float pole může být poskytnuto do MonoBehavioura, které doposud generovalo rovný mesh. V jejím kódu určíme vrcholu jeho výšku na základě noise mapy.

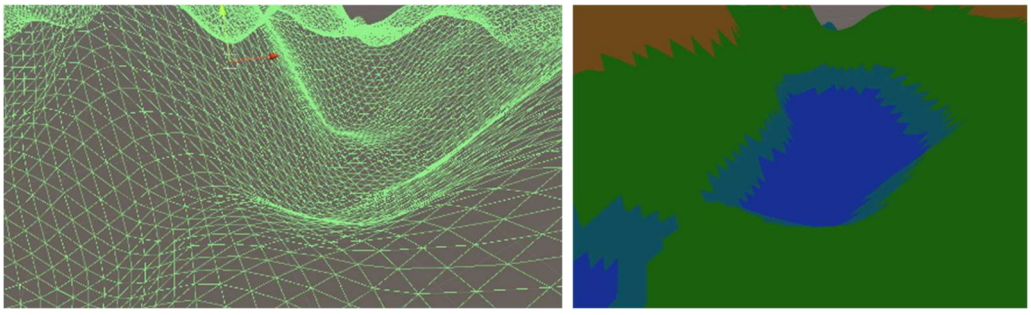


*Obrázek 96 - Rovný mesh*



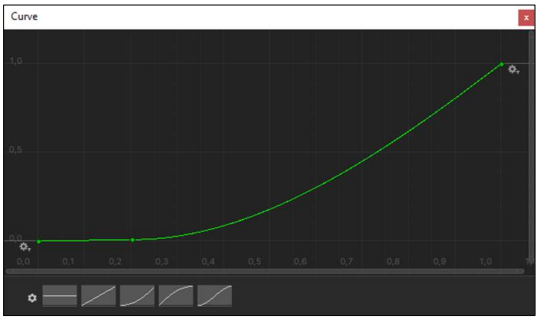
*Obrázek 97 - Noise změněný mesh*

Pro lepší zpracování výsledků šumu byla také přidána hodnota určující ostrost změn pro určité meze výšky. Díky tomu lze změnit chování pro některé hodnoty a tím například vytvoříme rovnou plochu na místě „vody“ nebo prudší vrcholy hor.



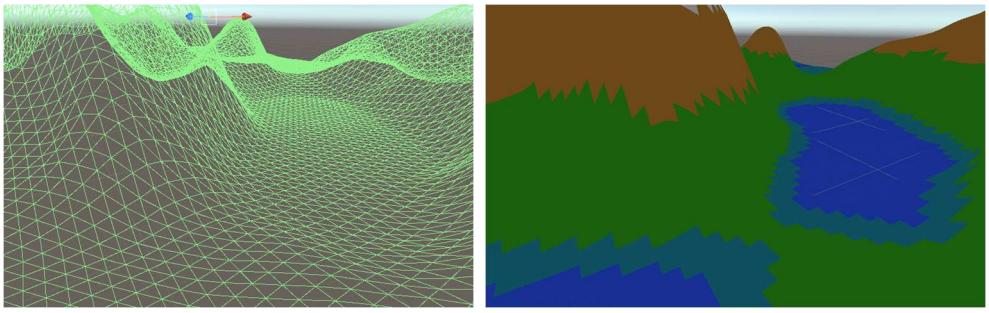
*Obrázek 98 - Nehlídaný mesh*

Pro určení ostrostí vytvoříme proměnnou typu `AnimationCurve`. Jedná se o standardní součást Unity, která je využívána pro animace, nicméně lze ji využít kdekoliv, kde je potřeba funkce přeměňující hodnotu jedné osy na druhou.




*Obrázek 99 - Animační křivka*

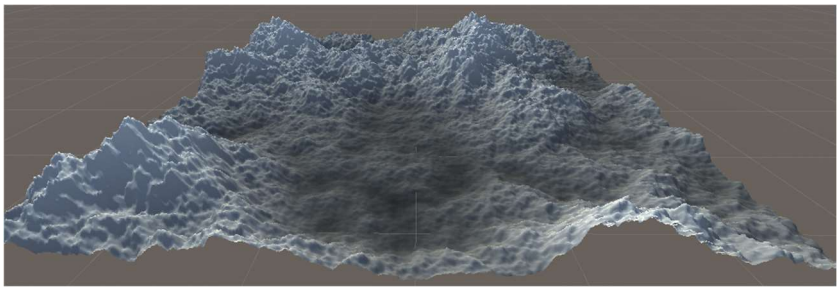
Pro nás tato funkce přepočte hodnoty osy X, která je hodnota pozice v noise mapě na hodnotu, o kterou skutečně posuneme mesh.



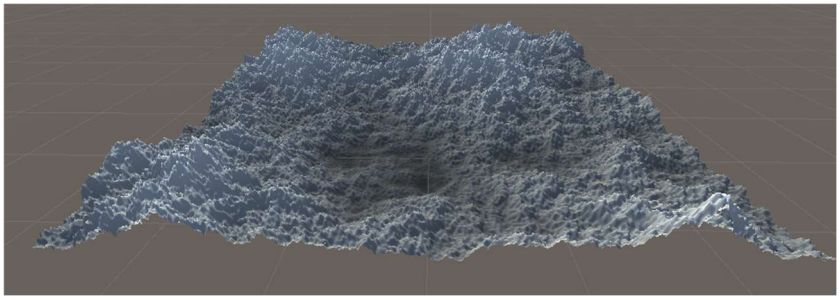
*Obrázek 100 - Funkcí hlídaný mesh*

## 4.3 Generování detailů

Mesh vytvořený na základě předchozích kroků již silně připomíná terén. Nicméně u něho lze chybným nastavením snadno docílit velmi nerealistického vzhledu.



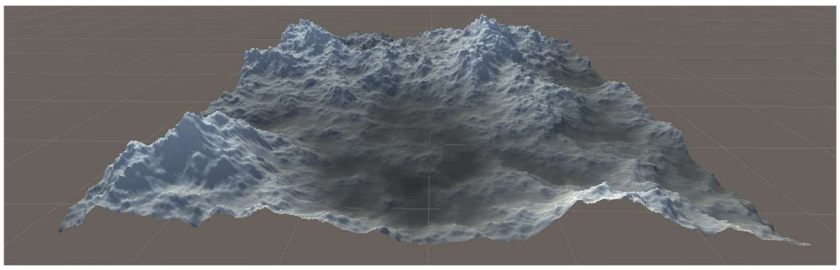
*Obrázek 101 - Příklad terénu 1*



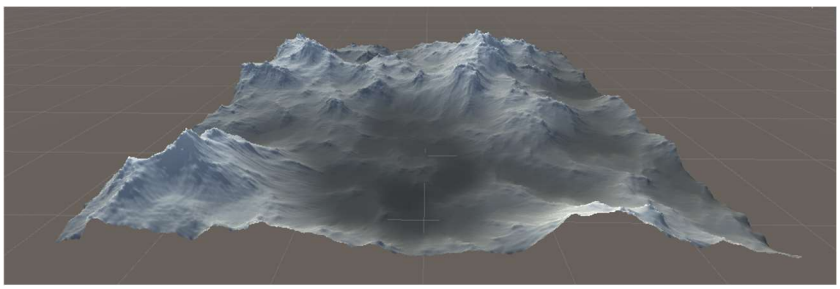
*Obrázek 102 - Příklad terénu 2*

Na základě teoretického popisu z předchozích kapitol bylo testováno rozdílné fungování jednotlivých metod. Na základě toho byla vytvořena metoda spustitelná na již existující prostor. Tato metoda terén pozmění na základě simulační metody hydraulické eroze, kterou jsme popsali v kapitole 2.3.1. Eroze dostane jako vstupní parametr datovou mapu původního terénu a na základě svého nastavení provede přepočtení těchto výšek.

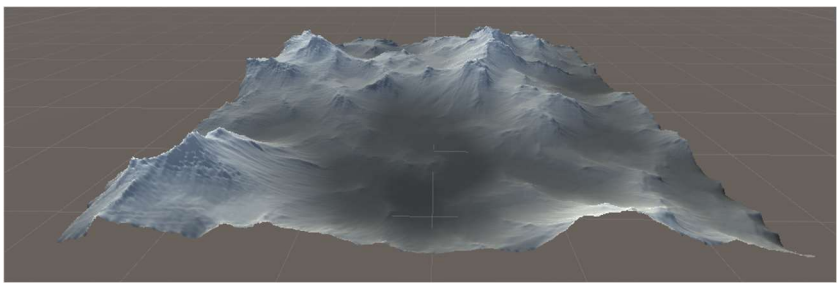
Tato eroze je založena na Compute Shaderu. Toho je využito z důvodu velké paralelizace eroze. Každá kapka je simulována nezávisle na ostatních, díky čemuž je možnost využití GPU vhodná. Compute Shader pracuje pouze nad daty, a to nezávisle na všem ostatním. Jelikož shadery lze automaticky spustit ve velkém počtu, tato data by měla být společná pro všechny výpočty. Z toho důvodu musí být proměnné, které jsou rozdílné v jednotlivých běžích tohoto shaderu, vytvořeny až v něm. Shader je napsán tak, aby simuloval dříve popsaný postup, a tím reprezentoval jednu kapku. Proto jsou mu předány všechny informace ohledně nastavení eroze a data terénu, nad kterým pracuje. Každá verze shaderu si poté sama vytvoří proměnnou určující její současnou pozici.



*Obrázek 103 - Eroze příklad 1 | 100 000 kapek*



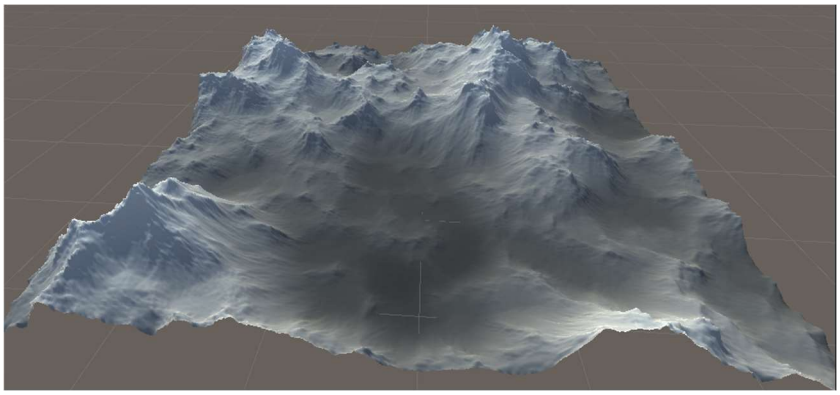
*Obrázek 104 - Eroze příklad 1 | 500 000 kapek*



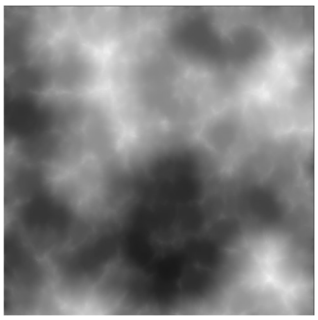
*Obrázek 105 - Eroze příklad 1 | 1 000 000 kapek*

## 4.4 Generování materiálu

Terén v ukázkách byl texturován pomoci noise mapy, která byla použita pro generování výšky.

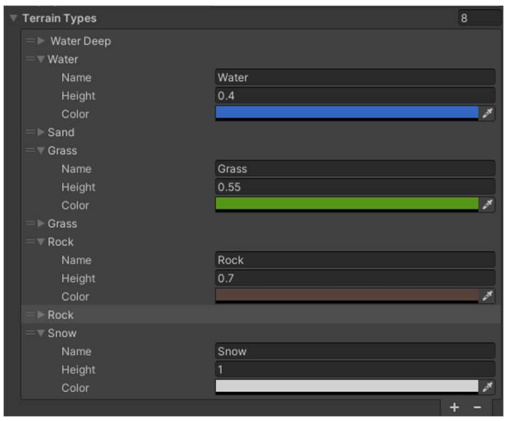


*Obrázek 106 - Height map Texturovaný terén*



*Obrázek 107 - Height map Textura*

Skutečný terén ale není černobílý. Z toho důvodu je vytvořen generátor textur, který na základě barvy ze spektra černé a bílé v hodnotách 0-1, přiřadí jinou barvu textury.

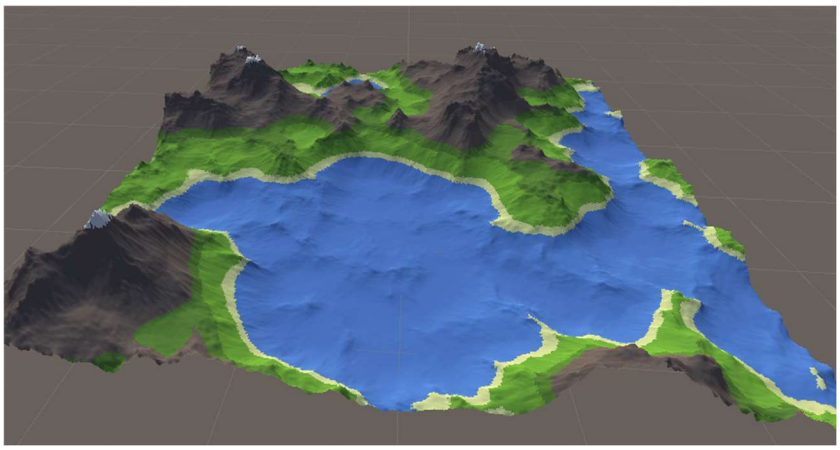



*Obrázek 108 - Typy terénu*

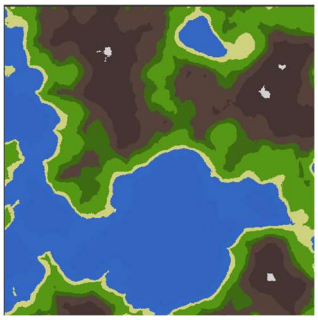
Pro každý pixel v textuře je nalezeno rozmezí barvy, do které spadá.

```
float currentHeight = noiseMap[x, y];
for (int i = 0; i < terrainTypes.Length; i++)
    if (currentHeight <= terrainTypes[i].height)
    {
        colorTexture[y * terrainWidth + x] = terrainTypes[i].color;
        break;
    }
```

*Obrázek 109 - Přiřazení barev*

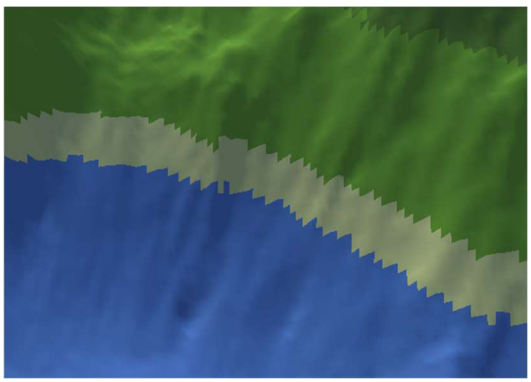


*Obrázek 110 - Barvami Texturovaný terén*



*Obrázek 111 - Barvami vytvořená textura*

Jak způsob texturace height mapou, tak i texturace pomoci barev, má jednu nevýhodu. Barvy jsou přiřazovány jednotlivým polygonům. Z toho důvodu vznikají velmi ostré hrany v textuře.



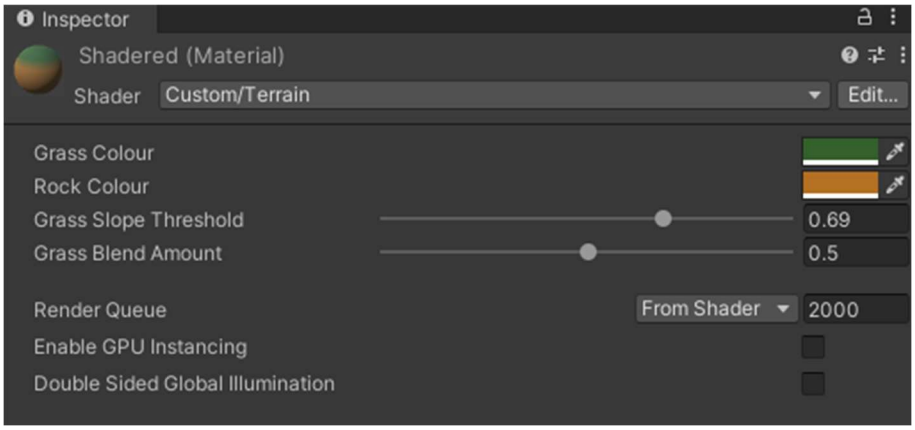
*Obrázek 112 - Ostré zlomy textury*

Tento problém lze vyřešit vlastním shaderem. Tomu poskytneme vlastnosti barev a proměnných, které určí jeho chování. Na jejich základě se vypočte skutečná barva pixelu a tím dosáhneme postupného přechodu místo ostré hrany.

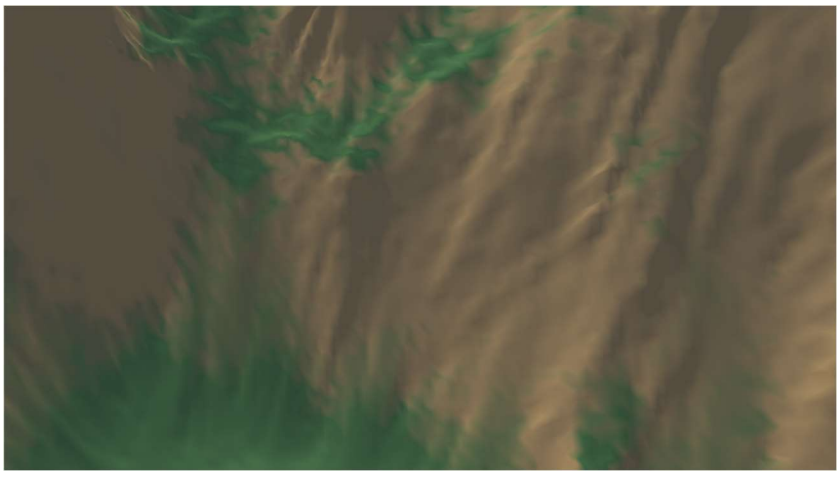
Shader má v sobě hlavní metodu surf určující způsob zobrazování každého pixelu. Vstupem je ‚IN' proměnná informující o datech prostředí, například pozice pixelu na meshi. Druhým je proměnná ‚o', která obsahuje odkaz na výstup shaderu. My vytváříme texturu barvy, tedy vytvoříme o.Albedo texturu, definovanou jako float3 hodnota.

o.Albedo = float3(R,G,B); v hodnotách 0-1

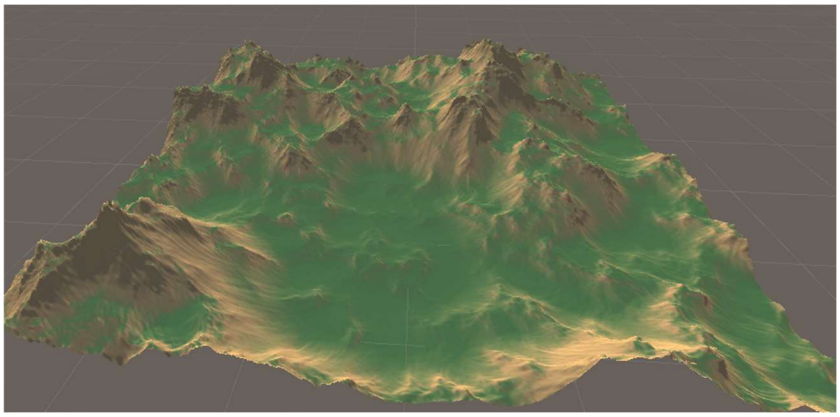
Shader lze poté využít ve vlastním materiálu. V něm lze přistupovat k proměnným, které byly v shaderu vytvořeny a jejich nastavením lze docílit různých variant plynulého přechodu mezi texturami.




*Obrázek 113 - Vlastní materiál*

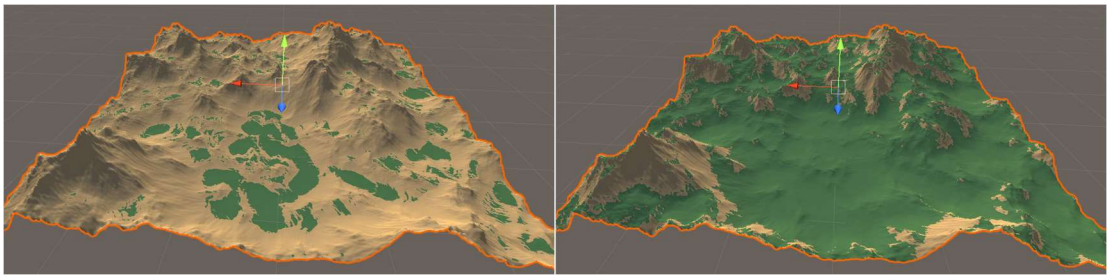


*Obrázek 114 - Hladké změny textur*

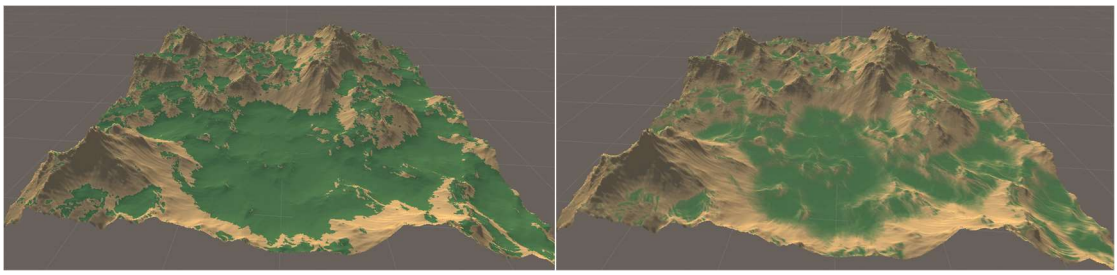


*Obrázek 115 - Shaded terén*

Různá nastavení určují rychlost změn a úhel povrchu, na kterém se textury „udrží“.



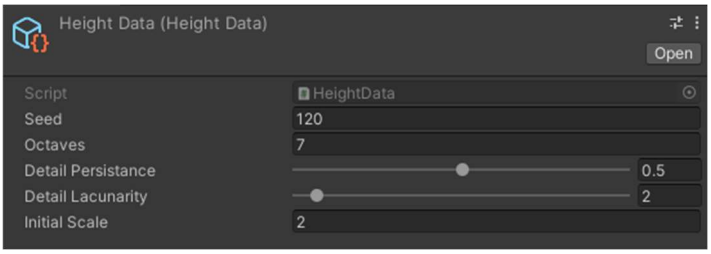
*Obrázek 116- Shader - Rozdílný úhel*



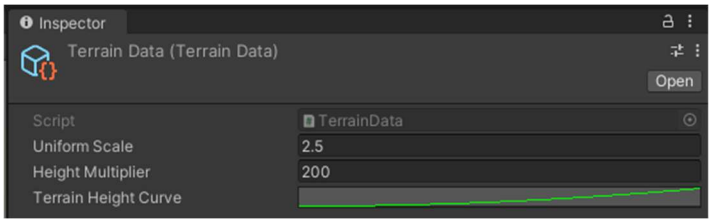
*Obrázek 117- Shader - Rozdílné míchání barev*

## 4.5 Parametrizace systému

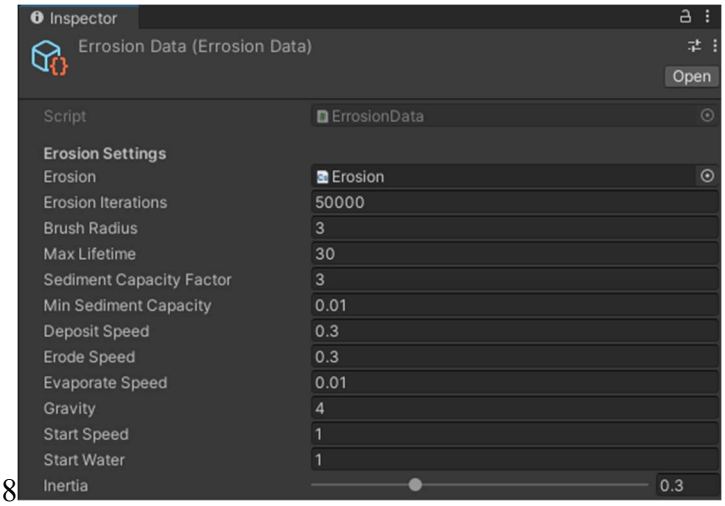
Systém je navržen na principu oddělených komponent. Díky tomu je poměrně snadné využít jejich data a díky serializaci je uchovat. Na to využijeme ScriptableObject.




*Obrázek 118 - Height Data*

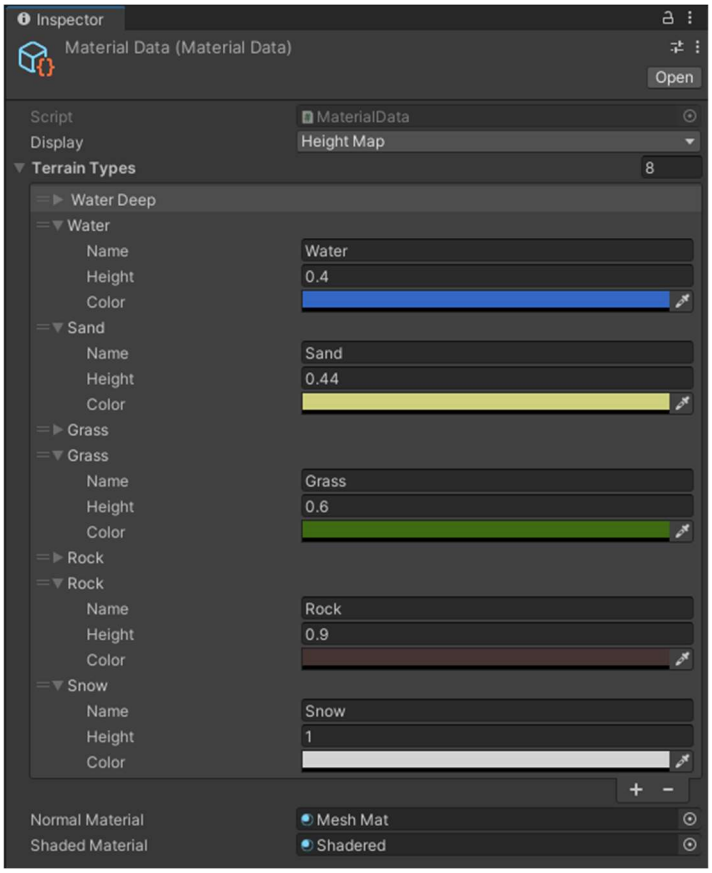



*Obrázek 119 - Terrain Data*




8

*Obrázek 120 - Errosion Data*

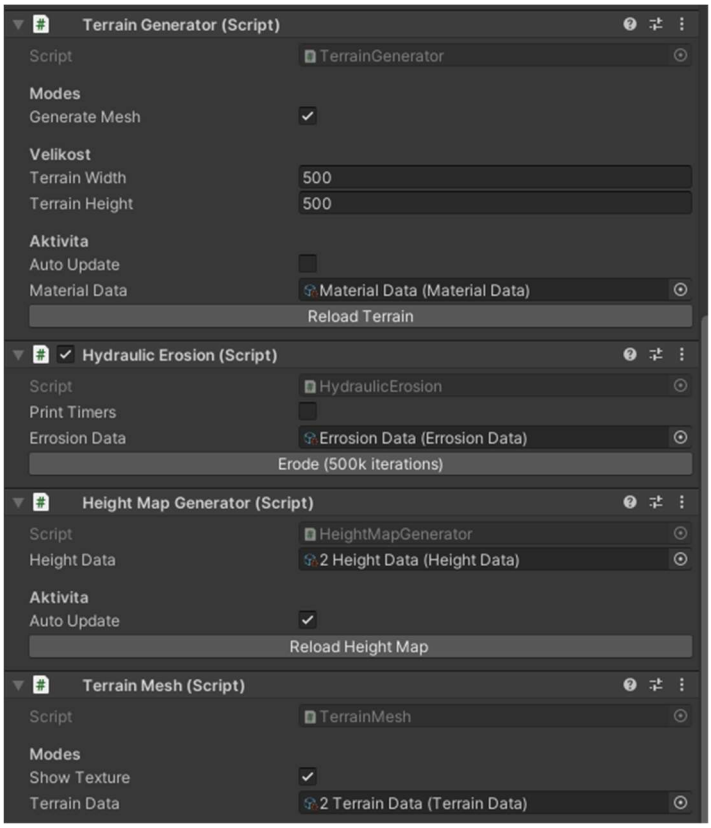



*Obrázek 121 - Material Data*

Takto můžeme uchovat různá nastavení, kdy každé obsahuje svá data. Pokud u nich již došlo k vypočtení a data se nezmění, lze výsledky v nich uložené znovu využít a tím šetřit výpočetní výkon.  Například při vytváření různých variant eroze není třeba přepočítávat výchozí mapu šumu.


*Obrázek 122 - Varianty Dat*

Scriptable objekty jsou pouze „zapojeny“ do jednotlivých MonoBehaviour řešících procesy.




*Obrázek 123 - MonoBehavioura*

## 4.6 Vyhodnocení výsledků

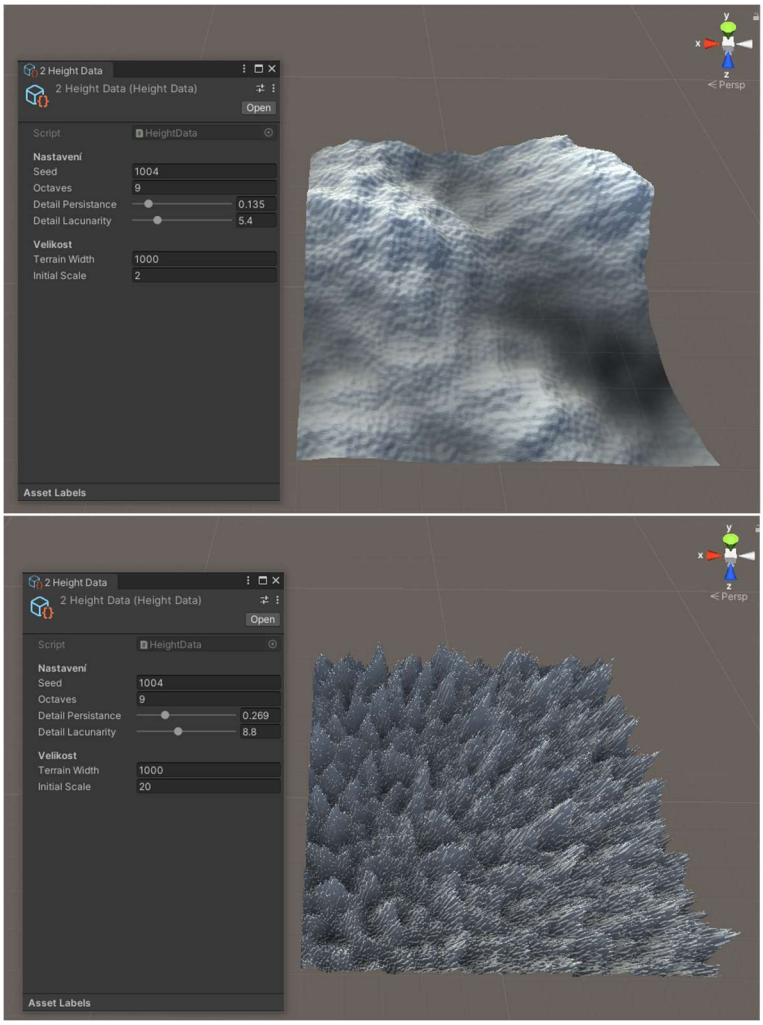
V této diplomové práci byl vyvinut nástroj, který bez jakýchkoliv jiných doplňků vytvoří všechny potřebné součásti procedurálně generovaného terénu.

Nástroj umožňuje:

- Perlin noise mapu – height mapa,
- Texturační mapu, vč. mapy pro vlastní „blend“ shader,

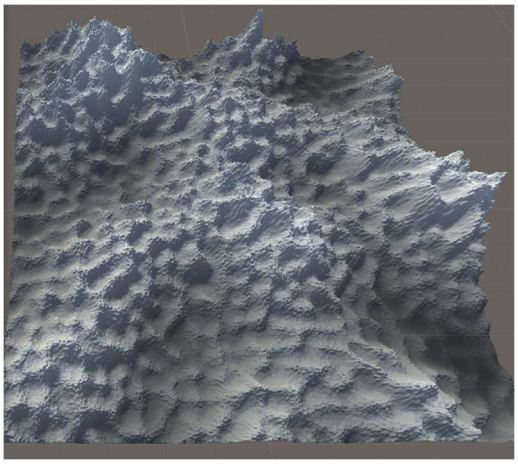
- Hydraulic errosion – zpracována pomoci compute shaderu (GPU),
- Při využití uvnitř Unity (ne jako standalone generátor) lze výsledný terén dále zpracovávat jinými Unity nástroji,
- Export jako .obj 3D objekt a využít jej v jakémkoliv 3D modeling software.

Na základě testů prováděných během implementace, i po jejím kompletním dokončení bylo ověřeno, že chování různých nastavení generovaných map vytváří různě realistické výsledky. Některé z nastavení však vytváří terén, který realistický není a bylo by proto vhodné parametry omezit.



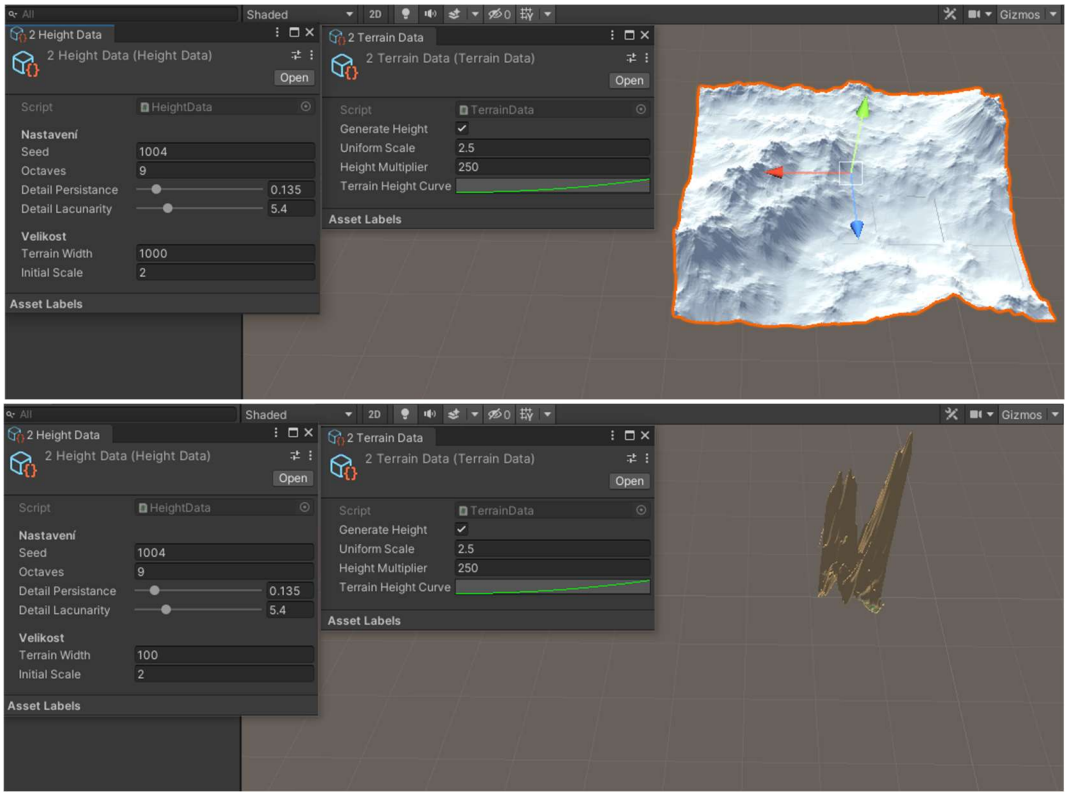

*Obrázek 124 - Šum způsobený persistance*

I velmi „šumivě“ vypadající terén může však po dostatečném množství erozních cyklů vypadat jako terén skutečně se nacházející v přírodě.



*Obrázek 125 - Terén s velkým šumem po vyhlazení*

Je také nutno zmínit vztah mezi násobitelem výšky a šířkou generovaného terénu.

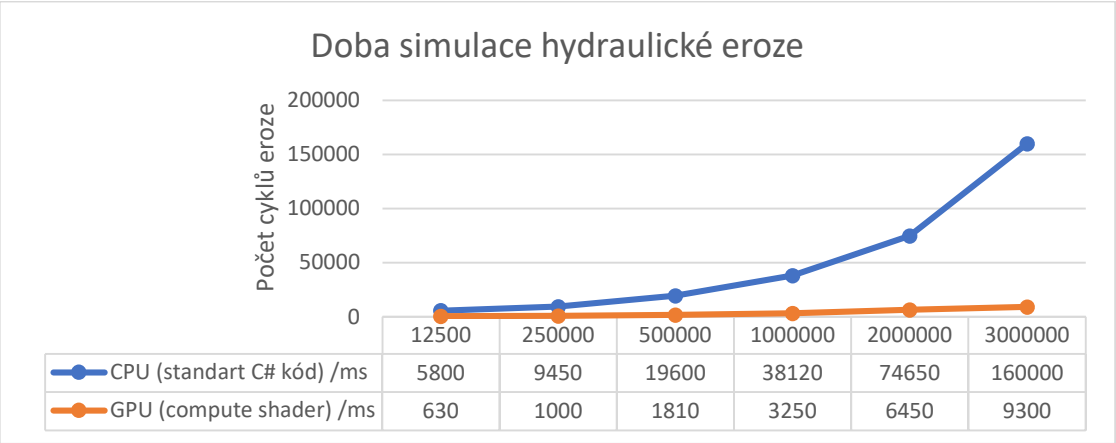



*Obrázek 126 - Terény s rozdílým násobičem výšky*

Výsledek poskytovaný nástrojem vyvinutým v této práci je tedy extrémně závislý na zadaných parametrech. Lze díky němu vytvořit velmi rozmanitý stylu terénu, který s trochou ladění parametrů odpovídá skutečnému prostoru v přírodě.

### 4.6.1 Výkon

Z výkonnostního hlediska se využití compute shaderu pro náročné výpočty jevilo jako velmi vhodné. Byl proveden zátěžový test, kde na stejném nastavení terénu byla provedena hydraulická eroze o určitém počtu cyklů a porovnána náročnost na výkon. Následující tabulka je vytvořená z průměru 100 generování na stejném terénu.

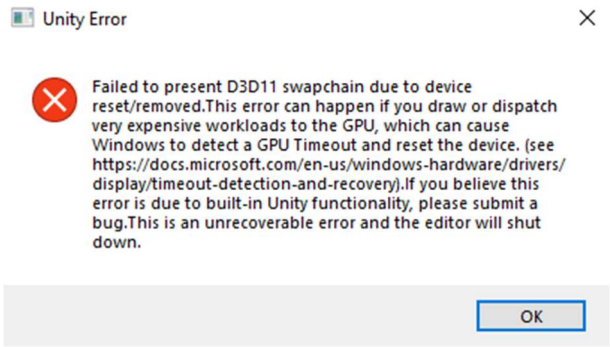



| | 12500 | 250000 | 500000 | 1000000 | 2000000 | 3000000 |
|---|---|---|---|---|---|---|
| CPU (standart C# kód) /ms | 5800 | 9450 | 19600 | 38120 | 74650 | 160000 |
| GPU (compute shader) /ms | 630 | 1000 | 1810 | 3250 | 6450 | 9300 |

*Obrázek 127 - Tabulka – Doba simulace hydraulické eroze*

Výsledky ukazují, že GPU i CPU udržovalo zhruba lineární zvyšování zátěže, nicméně na CPU docházelo k prudšímu nárustu, pravděpodobně z důvodu overheadu způsobeného nutností načítání dat. Poměr trvání výpočtu byl při 125 000 iteracích zhruba 1:9, při 3 000 000 iterací se rozdíl zvětšil až na 1:17. Compute shader tedy více než splnil požadované zlepšení výkonu.

Při větším počtu kroků došlo ale u GPU výpočtu k chybě.




*Obrázek 128 - Error paměti při hodně iteracích*

Tato chyba je způsobena nedostatkem paměti pro výpočet. Protože je celý výpočet vytvářen na GPU paralelně, je potřeba uchovávat data jednotlivých kroků pro každý výpočet a tím narůstá i paměťový problém. Možným rozšířením a optimalizací by mohlo být budoucí doplnění mezikroků pro výpočet. Tedy stanovení limitu, kolik kroků eroze

se provádí současně, vytvoření nové textury po tomto částečném výpočtu a následné počítání dalších erozí na základě průběžného terénu.
Tyto testy byly prováděny na terénu o 1000×1000 vrcholech. Vznikl tak terén o 1 000 000 vrcholech a 1 996 002 unikátních polygonů. Datová velikost výsledného terénu byla v rozmezí 20-55MB dle členitosti terénu a velikost textur byla v průměru 5MB. S tím, že všechny tyto velikost jsou lineárně proporční vůči velikosti terénu.

### 4.6.2 Limity

Větším problémem jsou data spotřebovávána systémem během generování textur. Tato data narůstala sice také lineárně, avšak díky množství informací potřebné především u hydraulické eroze, vyžadovali zhruba 120násobek prostoru, než byla velikost výsledného objektu. Tedy pro 1000×1000 terén bylo využito mezi 2,5GB a 5GB paměti. Výsledek projektu lze díky principům Unity spouštět i jako webový software, nicméně tyto paměťové potřeby silně omezují takovéto využití.

# 5 Závěr

V práci byly popsány teoretické postupy generování realistického terénu a byl a vyvinut systém, který je schopný samostatně vytvářet 3D terén. Na uměle vygenerovaném terénu aplikuje přírodou inspirované techniky tak, aby jeho vzhled lépe napodoboval terénní struktury existující v reálné přírodě.

Terén je vytvářen za pomoci generování sérií náhodných šumů. Tímto vzniklá černobílá textura reprezentuje výškovou strukturu základního terénu, který je poté doplněn následujícími metodami o dodatečné detaily.

Hlavní využitou technikou je hydraulická eroze, jejíž princip je založen na simulování velkého množství „kapek“ vody. Ty jsou schopny přemístit část „sedimentu“ v terénu a tím změnit výšky jeho jednotlivých polygonů.

Dále byly využity dvě metody, které jsou schopny přidávat nový detail do terénu. Tyto techniky jsou založeny na napodobení struktury řeky a na pohybu částí terénu neboli simulaci tektonických desek.

Kombinace těchto metod by bylo možno využít i opakovaně, kde na základním terénu dojde k vytvoření detailu pomoci hydraulické eroze. Výsledný terén je přegenerován pomocí pohybu tektonických desek a na nově vzniklé ploše můžeme opět simulovat erozi. Tento opakovaný postup byl v práci testován a jeho výsledky vypadaly velmi realisticky, nicméně nelze opakovaně využít generování řek, jelikož vzniklý terén poté ztrácel většinu souvislostí a začal napodobovat pouze náhodnou mapu šumu propojenou čarami.

Systém je založen na parametrizovaném vlivu objektů na terén. Parametry jsou v momentu generováni již předurčeny. Výsledkem je pak částečně opakovatelná možnost generování, ale omezuje se tím rozmanitost prostoru. Na základě myšlenky vzniklé během finálního testování je možnost dalšího rozšíření této práce, které by fungovalo s proměnlivými parametry, které by bylo možné měnit i v průběhu generování.

# 6 Summary and keywords

This thesis conducts a detailed analysis of methods that can be used to generate terrain in 3D software. Described procedures are ranged from methods for generating random terrain structures, to methods inspired by nature.
Combination of theses principles was implemented and tested as a standalone software tool. Focus of this tool is to allow for parametrized creation of terrain, that can be hand altered later.

Some parts of this tool were created in different ways (CPU and GPU based programming). Their quality and speed were also tested, and their results compared.

Key words: procedural, terrain, generation, hydraulic erosion, unity, 3D graphics

# 7 Seznam literatury


[1] P. Getreuer, „A Survey of Gaussian Convolution Algorithms,“ 2013.

[2] K. Perlin, An image synthesizer, ACM SIGGRAPH Computer Graphics, 1985.

[3] B. BENEŠ, V. TĚŠÍNSKÝ, J. HORNYŠ a S. K. BHATIA, Hydraulic erosion. Computer Animation and Virtual Worlds.

[4] B. BENEŠ, „Real-time erosion using shallowwater simulation,“ v *The 4th Workshop on Virtual Reality Interactions and Physical Simulation*, Vriphys, 2007.

[5] J.-D. Genevaux, E. Galin, E. Guérin, A. Peytavie a B. Benes, „Terrain Generation Using Procedural Models Based on Hydrology,“ 2013. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01339224/document.

[6] J. ŽÁRA, Moderní počítačová grafika, Brno: Brno: Computer Press, 2004.

[7] UNITY TECHNOLOGIES, „Unity Interface,“ [Online]. Available: https://docs.unity3d.com/Manual/UsingTheEditor.html. [Přístup získán 20 únor 2022].

[8] UNITY TECHNOLOGIES, „Scripting Backends,“ [Online]. Available: https://docs.unity3d.com/Manual/scripting-backends.html. [Přístup získán 20 únor 2022].

[9] UNITY TECHNOLOGIES, „Speciální třídy,“ [Online]. Available: https://docs.unity3d.com/Manual/ScriptingImportantClasses.html. [Přístup získán 20 únor 2022].

[10] UNITY TECHNOLOGIES, „Serializace,“ [Online]. Available: https://docs.unity3d.com/Manual/script-Serialization.html. [Přístup získán 20 únor 2022].

[11] UNITY TECHNOLOGIES, „Mono Behaviour,“ [Online]. Available: https://docs.unity3d.com/Manual/class-MonoBehaviour.html. [Přístup získán 20 únor 2022].

[12] UNITY TECHNOLOGIES, „Pořadí činností,“ [Online]. Available: https://docs.unity3d.com/Manual/ExecutionOrder.html. [Přístup získán 20 únor 2022].

[13] UNITY TECHNOLOGIES, „Events,“ [Online]. Available: https://docs.unity3d.com/Manual/EventFunctions.html. [Přístup získán 20 únor 2022].

[14] T. Wasson, „Rigidbodies with Time.timeScale,“ [Online]. Available: https://forum.unity.com/threads/rigidbodies-experience-huge-changes-invelocity-if-they-are-colliding-when-i-change-time-timescale.275707/. [Přístup získán 20 únor 2022].

[15] UNITY TECHNOLOGIES, „Coroutines,“ [Online]. Available: https://docs.unity3d.com/Manual/Coroutines.html. [Přístup získán 20 únor 2022].

[16] UNITY TECHNOLOGIES, „Scriptable Object,“ [Online]. Available: https://docs.unity3d.com/Manual/class-ScriptableObject.html. [Přístup získán 20 únor 2022].

[17] UNITY TECHNOLOGIES, „Extending The Editor,“ [Online]. Available: https://docs.unity3d.com/Manual/ExtendingTheEditor.html. [Přístup získán 20 únor 2022].

[18] UNITY TECHNOLOGIES, „Attributes,“ [Online]. Available: https://docs.unity3d.com/Manual/Attributes.html. [Přístup získán 20 únor 2022].

[19] UNITY TECHNOLOGIES, „Property Drawer,“ [Online]. Available: https://docs.unity3d.com/Manual/editor-PropertyDrawers.html. [Přístup získán 20 únor 2022].

[20] UNITY TECHNOLOGIES, „Inspector,“ [Online]. Available: https://docs.unity3d.com/Manual/InspectorOptions.html. [Přístup získán 20 únor 2022].

[21] UNITY TECHNOLOGIES, „Editor,“ [Online]. Available: https://docs.unity3d.com/ScriptReference/Editor.html. [Přístup získán 20 únor 2022].

[22] UNITY TECHNOLOGIES, „Custom Editors,“ [Online]. Available: https://docs.unity3d.com/Manual/editor-CustomEditors.html. [Přístup získán 20 únor 2022].

[23] UNITY TECHNOLOGIES, „Event System,“ [Online]. Available: https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/EventSystem.html. [Přístup získán 20 únor 2022].

[24] UNITY TECHNOLOGIES, „Mouse Events,“ [Online]. Available: https://docs.unity3d.com/Manual/UIE-Mouse-Events.html. [Přístup získán 20 únor 2022].

[25] UNITY TECHNOLOGIES, „Mesh,“ [Online]. Available: https://docs.unity3d.com/Manual/mesh.html. [Přístup získán 20 únor 2022].

[26] Atlassian Confluence, „Culling explained,“ [Online]. Available: https://docs.cryengine.com/display/SDKDOC4/Culling+Explained. [Přístup získán 20 únor 2022].

[27] UNITY TECHNOLOGIES, „Post Procesing,“ [Online]. Available: https://docs.unity3d.com/Manual/PostProcessingOverview.html. [Přístup získán 20 únor 2022].

[28] UNITY TECHNOLOGIES, „Material,“ [Online]. Available: https://docs.unity3d.com/Manual/Materials.html. [Přístup získán 20 únor 2022].

[29] UNITY TECHNOLOGIES, „Textures,“ [Online]. Available: https://docs.unity3d.com/Manual/Textures.html. [Přístup získán 20 únor 2022].

[30] UNITY TECHNOLOGIES, „Shaders,“ [Online]. Available: https://docs.unity3d.com/Manual/Shaders.html. [Přístup získán 20 únor 2022].

[31] UNITY TECHNOLOGIES, „Fyzika,“ [Online]. Available: https://docs.unity3d.com/Manual/PhysicsSection.html. [Přístup získán 20 únor 2022].

# 8 Seznam obrázků