



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
BRNO UNIVERSITY OF TECHNOLOGY

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**  
FACULTY OF INFORMATION TECHNOLOGY

**ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ**  
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

**GENEROVÁNÍ 2D MAP PRO POČÍTAČOVÉ HRY**  
2D MAP GENERATION FOR COMPUTER GAMES

**BAKALÁŘSKÁ PRÁCE**  
BACHELOR'S THESIS

**AUTOR PRÁCE**  
AUTHOR

**KRYŠTOF GLOS**

**VEDOUCÍ PRÁCE**  
SUPERVISOR

**Ing. MICHAL MATÝŠEK**

**BRNO 2021**

## Abstrakt

Tato práce se zaměřuje na jednotlivé metody procedurálního generování z hlediska herního průmyslu, zejména v oblasti vytváření map. Cílem této práce, je vytvoření hry žánru Colony-sim, využívající procedurální generování obsahu.

Zaměřil jsem se na analýzu různých způsobů vytváření obsahu a rozbor metod pomocí kterých lze automatizovaně generovat herní prostředí.

Zvolenou metodou pro tvoření herní oblasti byl Perlinův šum, s následným využitím výškových map. V práci představuji algoritmus, jenž na základě vstupních parametrů generuje funkční, unikátní a neopakující se mapy, což zajišťuje opakovatelnou hratelnost. Klíčovým výstupem této práce, je hra implementující procedurální algoritmus generující mapy, který je aplikovatelný v širokém spektru 2D her.

## Abstract

This work focuses on individual methods of procedural generation in the context of the gaming industry, particularly in the realm of map creation. The aim of this study is to develop a Colony-sim genre game utilizing procedural content generation.

I have concentrated on analyzing various methods of content creation and scrutinizing techniques through which gaming environments can be automatedly generated.

The chosen method for crafting the gaming area was Perlin noise, coupled with the subsequent utilization of height maps. In this work, I present an algorithm that, based on input parameters, generates functional, unique, and non-repeating maps, ensuring repetitive playability. The key outcome of this study is the implementation of a game incorporating a procedural algorithm for map generation, applicable across a broad spectrum of 2D games.

## Klíčová slova

Procedurální generování obsahu, 2D mapy, 2D hry, počítačové hry, hry o přežití

## Keywords

Procedural content generation, 2D maps, 2D games, computer games, survival games

## Citace

GLOS, Kryštof. *Generování 2D map pro počítačové hry*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Matýšek

# Generování 2D map pro počítačové hry

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....  
Kryštof Glos  
2. května 2024

## Poděkování

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant apod.).

# Obsah

<b>1 Enginy na vývoj her</b>	<b>4</b>
1.1 Construct 2 . . . . .	4
1.2 Unreal engine . . . . .	4
1.3 Unity . . . . .	4
<b>2 Vytváření obsahu v herním světě</b>	<b>6</b>
2.1 Metody generování herního obsahu . . . . .	6
2.1.1 Mechanické generování obsahu . . . . .	7
2.1.2 Procedurální generování obsahu . . . . .	7
2.2 Procedurální generování v herním průmyslu . . . . .	9
2.2.1 Text . . . . .	9
2.2.2 Krajina a úrovně . . . . .	10
2.2.3 Zvuky a hudba . . . . .	10
2.2.4 Textury . . . . .	11
<b>3 Metody procedurálního generování</b>	<b>12</b>
3.1 Celulární automaty pro procedurální generování . . . . .	12
3.2 L-systémy . . . . .	13
3.2.1 Geometrie pomocí L-systémů . . . . .	15
3.3 Šumy . . . . .	16
3.3.1 Definice šumu . . . . .	16
3.3.2 Gradientní mřížkové šumy . . . . .	17
3.3.3 Explicitní šumy . . . . .	18
3.3.4 Řídké konvoluční šumy . . . . .	21
3.4 Výškové mapy . . . . .	23
<b>4 Návrh řešení</b>	<b>25</b>
4.1 Vybrané technologie . . . . .	25
4.2 Návrh procedurálního generování oblastí . . . . .	26
4.2.1 Zvolený šum . . . . .	26
4.2.2 Algoritmus generování mapy pomocí Perlinova šumu . . . . .	27
4.2.3 Algoritmus Wave Function Collapse . . . . .	28
4.3 Návrh hry . . . . .	29
4.3.1 Grafika . . . . .	30
4.3.2 Jednoduchá umělá inteligence nepřátel . . . . .	30
4.3.3 NavMesh funkcionality . . . . .	30
4.3.4 Ekonomika hry . . . . .	31
4.4 Struktura projektu . . . . .	33

<b>5 Implementace</b>	<b>34</b>
5.1 Implementace Perlinova šumu . . . . .	34
5.1.1 Skript pro inicializaci a nastavení šumu . . . . .	35
5.1.2 Třída Noise . . . . .	36
5.2 Wave function collapse . . . . .	37
5.2.1 Skript WFCTile . . . . .	37
5.3 Hráčovy jednotky . . . . .	39
5.3.1 Řízení a pohyb jednotek . . . . .	39
5.3.2 Ostatní třídy . . . . .	40
5.3.3 Interakce a funkcionalita . . . . .	40
5.4 Umělá inteligence nepřátel . . . . .	41
5.4.1 Detekce hráče . . . . .	42
5.4.2 Pronásledování hráče . . . . .	42
5.5 Dynamický systém stavění . . . . .	43
5.5.1 Vytváření stavebního menu . . . . .	44
5.5.2 Používání stavebního menu . . . . .	44
5.6 Náhodné události . . . . .	45
5.6.1 Třída EventManager . . . . .	46
5.6.2 Rozhraní IGameEvent . . . . .	47
<b>Literatura</b>	<b>48</b>

# Úvod

Procedurálně vytvářený obsah v herním průmyslu je velmi důležitou součástí her už po několik let. Mnoho her postavených na tomto principu se již prosadilo na trhu, a stále více se uplatňuje. Náhodné vytváření obsahu se používá například na tvoření herních map, včetně místnosti, skládání různých dopředu vytvořených místností tak, aby vznikla jedinečná mapa. Takto implementované hry mají výhodu v opakované hratelnosti a nepředvídatelnosti.

Cílem této bakalářské práce je návrh a vytvoření prototypu 2D hry v herním enginu Unity, založené na procedurálním generování herního obsahu. Samotný generační algoritmus pracuje s Perlinovým šumem a výškovými mapami, ty slouží k vytvoření elevace různých bodů na mapě.

Inspirací na téma hry je počítačová hra RimWorld, která se řadí do her typu Colony-sim. Jedná se o žánr, ve kterém hráč ovládá skupinu lidí (kolonii), kterou se snaží pomocí dobrého vedení dovést k nějakému danému cíli. V navržené hře je úkolem hráče vydržet co nejdéle nájezdy nepřátel a přečkat další náhodné události. Pohyb kolonistů využívá takzvané NavMesh agenty, kteří spolupracují s vytvořenou navigační sítí (NavMesh) popsané v kapitole 4.3.3.

Vytváření mapy je stěžejní částí celé hry. Určuje zdroje pro hráče, a tudíž částečně i obtížnost celé hry. Ve hře se objevují nepřátelé, kteří mají jednoduchou umělou inteligenci a jsou jednou z překážek, se kterou se hráč musí během hry vypořádávat. Implementace generačních algoritmů, AI nepřátel a dalšího je popsána v kapitole 5.

Hra je implementována v herním enginu Unity, který je v dnešní době jedním z nejvíce používaných vývojových nástrojů pro vývoj her. Mezi další oblíbená vývojová prostředí se řadí například, GameMaker, Unreal Engine, Godot. O vývoji v jednotlivých enginech pojednává kapitola 1.

Po dokončení implementační části bylo zahájeno experimentování s herními systémy a následovné uživatelské testování. Průběh a výsledky této části práce jsou dále popsány v kapitole ??.

# Kapitola 1

## Enginy na vývoj her

Herní engine představuje platformu složenou z interagujícího softwaru, který dohromady vytváří integrovaný celek a umožňuje spouštění samotných her. Herní engine se skládá z několika částí s přesně specifikovanou funkcionalitou: rendering, fyzika, síťování, zvuk atd. [31]

Platforem na vývoj her existuje mnoho, některými z nejvýznamnějších jsou: Unity, Construct 2, MonoGame, Unreal Engine nebo GameMaker Studio 2. Každý herní engine je v něčem jiný, a tudíž se hodí na jiné žánry, nebo styly her. Při rozhodování, který engine použít, se z hlediska vývojáře musí zohlednit více faktorů, například podporovaný programovací jazyk nebo platformu, na kterou je hra vyvíjena [43].

### 1.1 Construct 2

Tento engine dovoluje lidem, kteří nejsou programátori, vytvářet 2D hry. Používá drag and drop editor pro všechnu logiku založenou na událostech a chování. Může být rozšířen a skriptován pomocí JavaScriptu.

I když Construct tvrdí, že zveřejňuje hry na většině mobilních a desktopových platformách, jejich primární cíl je HTML5/JavaScript. Tudíž jakákoli verze, která není ve vyhledávači, je obsažena v DOM a obalovacím rozhraní umožňujícím použití JavaScriptu. Tato architektura obecně snižuje výkon [1].a

### 1.2 Unreal engine

Podporuje multiplatformní vydávání her, jmenovitě DirectX, OpenGL, nebo WebGL. Jedná se o engine, který je zdarma, ale pouze pro nekomerční užití a ve všech ostatních případech licencování softwaru za malé předplatné a licenční poplatek. Přesto, že původně byl vyvinut pro podporu *Unreal* her z první osoby a neměl tolík nástrojů jako Unity, vyrostl Unreal Engine do podoby velmi výkonného enginu schopného podporovat jakýkoli žánr her [1]. Unreal engine využívá programovacího jazyka C++.

### 1.3 Unity

Unity je multiplatformní herní engine podporující vývoj her na Windows, Linux i macOS. Vyvinula ho společnost Unity Technologies v roce 2005. Oproti jiným herním enginům podporuje vývoj her ve 2D, 3D, rozšířenou realitu (AR), nebo virtuální realitu (VR).

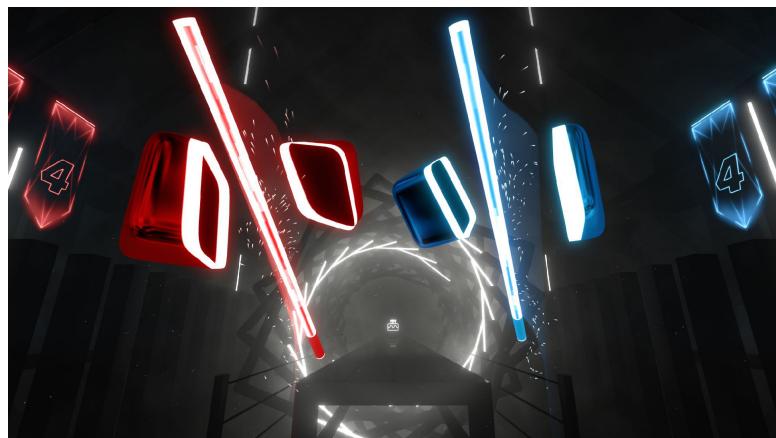
Unity má více plánů, které mohou vývojáři využívat, jedná se o personal, pro, enterprise a industry. Ze základu je pro všechny vývojáře zdarma k užívání, ale po překročení 100 000\$ za posledních dvanáct měsíců, je nutné pořídit si jeden z placených plánů. Tyto plány mají i jisté výhody, jako je třeba Havok Physics, přidávající robustní detekci kolizí a fyzikální simulace, technickou podporu, nebo prioritní frontu na zákaznickou podporu. [42]

Další výhodou je integrované fyzikální jádro PhysX, které pracuje v reálném čase a je vyvíjeno společností Nvidia. Toto jádro zahrnuje efektivní podporu pro multithreading a využívá akceleraci fyzikální simulace prostřednictvím GPU. Programování je v enginu zařízené pomocí jazyka C#.

Z Unity vzešlo mnoho oblíbených herních titulů. Řadí se mezi ně hra ve virtuální realitě Beat Saber 1.2, mobilní hra Pokémon Go, nebo multiplatformní Cuphead 1.1.



Obrázek 1.1: Ukázka hry Cuphead.



Obrázek 1.2: Ukázka hry Beat Saber.

## Kapitola 2

# Vytváření obsahu v herním světě

Ve světě herního vývoje existují dvě základní metody pro tvorbu herního obsahu. Jednou z nich je tradiční, nazývaná také mechanická, o této metodě je více napsáno v sekci 2.1.1. Tato metoda se vyznačuje ručním a přímým přístupem k tvorbě obsahu. Na rozdíl od algoritmických přístupů není nutné vytvářet složité algoritmy či programovat logiku pro generování obsahu. Namísto toho se obsah tvoří manuálně, často pomocí specializovaných nástrojů a editačních programů. Přestože je tento přístup časově náročnější a vyžaduje více práce, umožňuje větší kontrolu nad výsledným produktem a často poskytuje tvůrcům větší míru kreativity a individuálního projevu. Další možností vytváření obsahu je pomocí metod implementujících náhodné nebo také **procedurální generování obsahu**.

Hry, které mají pouze dvě dimenze se nazývají 2D (z anglického two dimensions). Je mnoho žánrů 2D her, RPG (role playing game) hry na hrdiny s příběhem, strategií, Co-op (kooperační), které jsou postavené na spolupráci více hráčů, survival (hry o přežití), colony-sim (z anglického colonization simulation), které mají simulovat kolonizaci ovládanou hráčem atd. Tato bakalářská práce se zabývá hrou žánru colony-sim. Je mnoho způsobů, jak vyvijet takovou hru.

### 2.1 Metody generování herního obsahu

V této sekci je srovnání mechanického generování herního obsahu s procedurálním přístupem a vysvětluje, který z nich je vhodnější v různých situacích. Dále se zaměřuje na různé metody procedurálního generování a faktory, které je třeba zohlednit při rozhodování o vhodnosti jejich použití oproti manuálnímu navrhování obsahu.

**Žánr hry** Při vývoji her různých žánrů je rozhodující, zda využít mechanického nebo procedurálního generování obsahu. Například u her z prvního pohledu (FPS), kde je důraz kláden především na akční prvky a souboje hráče proti protivníkům, není často nutné generovat úrovně procedurálně. V takových případech postačuje vytvořit omezený počet ručně navržených úrovní. Naopak u her, které zdůrazňují průzkum prostředí, sbírání surovin a přežití, může být procedurální generování klíčové pro dosažení rozmanitosti a nepředvídatelnosti herního prostředí.

**Opakování hratelnost** Při hodnocení vhodnosti použití procedurálního generování je důležité zohlednit očekávanou délku hratelnosti a opakovost herních zážitků. U her s dlouhodobou hratelností, kde je cílem zdokonalovat se a dosahovat stálé lepších výsledků, může být manuální navrhování úrovní preferovanou metodou. Naopak u

her s jednorázovými herními zážitky, kde hráč pravděpodobně úroveň nebo misi projde pouze jednou, může být procedurální generování vhodnější volbou pro zajištění rozmanitosti a obnovitelnosti herního obsahu.

**Aspekt designu hry** Záleží také na tom, zda design hry klade důraz na jednotlivé ručně navržené úrovně s pečlivě vyladěnými herními mechanikami a interakcemi, nebo zda je hlavním cílem dosažení rozmanitosti a dynamiky prostřednictvím procedurálního generování. V prvním případě je preferováno mechanické navrhování obsahu s důrazem na detailní ruční práci, zatímco v druhém případě může být procedurální generování klíčové pro dosažení požadovaného stupně variability a překvapení v herním světě.

### 2.1.1 Mechanické generování obsahu

Mechanický typ generování je jedním z nejobvyklejších tvoření obsahu ve hrách. Používá se převážně v žánrech, jako je RPG (Role Play Game), RTS (Real Time Strategy) a další, ve kterých pozice objektů a struktura mapy hraje velkou roli a bez lidské tvorby by nebylo dosaženo potřebných výsledků. Toto tvoření lze interpretovat jako proces u něhož se návrhář za pomoci různých nástrojů, které postupně aplikuje, snaží dosáhnout požadovaného výsledku. Jde tedy o metodu ručního vytváření obsahu kde designér, nebo grafik navrhuje a postupně vytváří úroveň, či jinou část hry tak, aby vyhovovala potřebám, at už se jedná o pozici stromu, nebo o to co hráči sdělují NPC.

Díky tomuto přístupu je eliminováno riziko vzniku nesrovnatostí ve výsledném herním prostředí, jako je nedostupnost části mapy nebo vznik absurdních herních situací. Další výhodou je zaručení, že výsledný herní obsah bude přesně odpovídat plánovaným specifikacím. Nicméně, je důležité zdůraznit, že tento přístup je časově náročný, zejména při vývoji rozsáhlejších her, kde je požadována opakování hratelnost a velké množství detailního obsahu.

### 2.1.2 Procedurální generování obsahu

Tento typ vytváření obsahu se používá ve více žánrech, ale asi nejznámější z hlediska generování map je Roguelike, kde každá nová hra má unikátní náhodnou mapu. Procedurální generování se ovšem nepoužívá pouze na generování map, ale také na vytváření objektů, jako jsou stromy, textury, animace, text a další. Procedurální generování obsahu není úplně to stejné, jako náhodné generování obsahu. Více do hloubky je tento typ modelování krajiny a textur rozebrán v kapitole [Procedurální generování](#).

V zásadě se jedná o proces obrácený, jako u mechanického generování obsahu. Uživatel sice stále definuje různé nástroje, které jsou použity pro vytváření obsahu, ale nikoli pro své vlastní použití, ale naopak je vytváří pro algoritmus. Uživatel dále určuje pravidla podle kterých se generátor musí řídit, tak aby se dobral kýzeného výsledku.



Obrázek 2.1: Příklad hry Roguelike žánru jménem Binding of Isaac, vpravo nahoře je vidět mapa dungeonu, která je procedurálně vygenerovaná.

### Příklad procedurálního generování vegetace

Simulace lesních prostředí má mnoho aplikací, a to od zábavní po výzkumné modelování. Pro lepší představu je uveden příklad od Newlandse a Zaunera. [30]

Program má funkci procedurálního generátoru lesů na mapě. Cílem tohoto programu je náhodně naskládat stromy s rozumnými rozestupy od sebe. Metody pro vegetaci se dají opět řadit jako procedurální a mechanické. Neprocedurální přístup k tvorbě obsahu poskytuje vývojářům širokou kontrolu nad celým procesem a umožňuje detailní modelování prostředí a objektů. Tato kontrola však přichází s cenou, neboť vyžaduje značné množství ručních úprav a intervencí. Vývojáři často stráví hodně času laděním a umisťováním jednotlivých prvků, jako je například ruční výsadba rostlin nebo úprava každého detailu prostředí. V případě procedurálního generování se jedná o modely, které jsou zkonstruovány algoritmicky a kontrolovaný skrze parametrické hodnoty bez nutnosti vyšší úrovni manuálního vstupu nebo specializace v daném oboru.

Jednou z nejvíce prozkoumanou modelační technikou postavenou na rekurzivních hierarchiích je koncept **L-systémů**. Existuje mnoho typů a rozšíření těchto systémů [36], včetně stochastických, parametrických, diferenciálních [37], citlivých na okolí [38], nebo otevřených [29].

Distribuce stromů na scéně lze udělat například pomocí simulace ekosystému. To vytváří více realistickou distribuci vegetace, než náhodné rozestavení, protože vznikají přirozená chování, jako je shlukování druhů a oblasti negativního růstu kolem stromů. [30] Jedná se o simulace, ve kterých každá rostlina žije vlastní život, je ovlivňována ostatními rostlinami a vnějšími podmínkami. [4]

Při inicializaci simulace se pro každý specifický druh stromu vytvoří určité množství stromů (pro výpočet slouží rovnice 2.1) ve věku  $a_i \in \langle 0, a_M \rangle$ , kde  $a_M$  je maximální věk

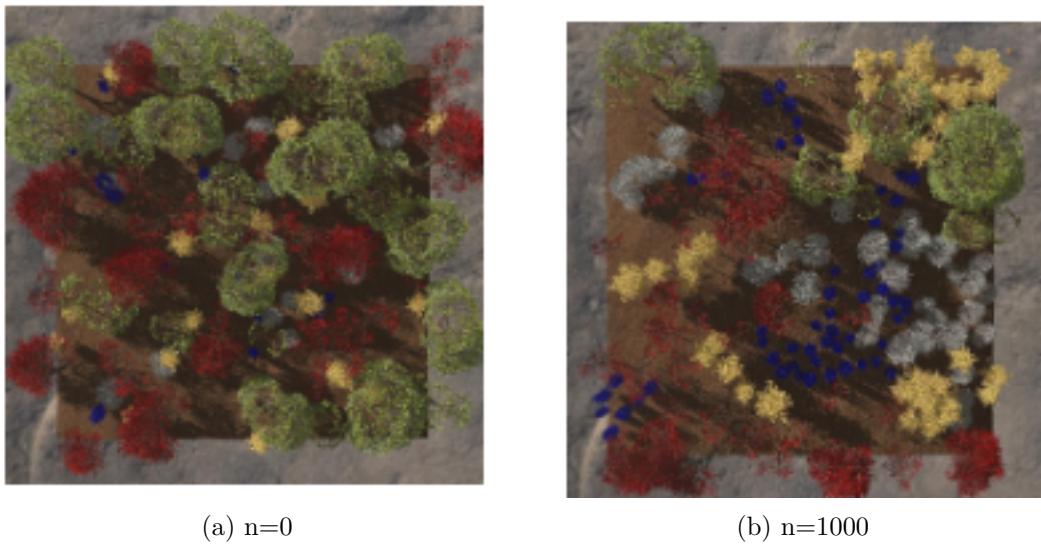
stromu pro daný druh a jsou náhodně rozestavěny po oblasti od největšího po nejmenší. [30]

$$n_I = \frac{d_0 \cdot \rho}{n_T} w^2 \quad (2.1)$$

Hodnota  $n_I$  představuje počet instancí k vytvoření,  $w$  je šířka scény,  $d_0$  původní hustota pro objekt a  $\rho$  je parametr hustoty jednotlivých objektů.

Po úspěšné inicializaci simulace běží  $N$  kroků, kde  $N_Y$  kroků utváří rok. Pro každý krok se děje následující:

1. Pokud je konec roku, všechny stromy vysejí počet nových rostlin, v kruhu okolo nich.
2. U každého páru stromů je rostlina s menší důležitostí odstraněna.
3. Stromy starší než jejich maximální věk, jsou považovány za mrtvé a odstraněny.
4. Všechny rostlinky rostou (jejich věk se zvýší o 1).



Obrázek 2.2: Vzhled ekosystému po inicializaci a rozložení vegetace po prvních 1000 iteracích. Je vidět že se vytváří shluky rostlinných druhů, tyto klastry zůstávají, pouze se mění jejich pozice.

## 2.2 Procedurální generování v herním průmyslu

Algoritmů na generování obsahu existuje mnoho, každý používá jiné nástroje, ale všechny se musí podrobovat pravidlům, která stanovuje programátor, a podle kterých se řídí. Je více způsobů a míst, kde se dá procedurální generování uplatnit, různé způsoby a důvody jsou popsány v této kapitole.

### 2.2.1 Text

Skoro všechny hry používají text. Z důvodu, že každá informace v textu musí odpovídat realitě, je nutné velké množství omezení pro generování. Například, když je v textu informace, že král je mrtvý [27], musí být toto tvrzení pravdivé.

Velkou výhodou procedurálního generování textu je vyprávění [8], takto vytvořené příběhy jsou často kreativnější a zajímavější, než ty, co by vytvořil člověk, nebo lidé mají sklonky psát příběhy, které již slyšeli, nebo ze svých zkušeností, což dost omezuje kreativitu.

### 2.2.2 Krajina a úrovně

Nejvíce obvyklý obsah, který se ve hrách generuje, a který je zároveň hlavním zaměřením této bakalářské práce, jsou krajiny a úrovně. Generování lokací, úrovní, nebo obsahu mapy lze jak u 2D her (generování krajiny demonstruje obrázek 2.3), tak u 3D her. Za úroveň, nebo oblast lze označovat otevřené (například krajina s lesy), i uzavřené prostranství, vnitřek budovy, nebo jeskyně.



Obrázek 2.3: Procedurálně vygenerovaná krajina s lesy a skálami

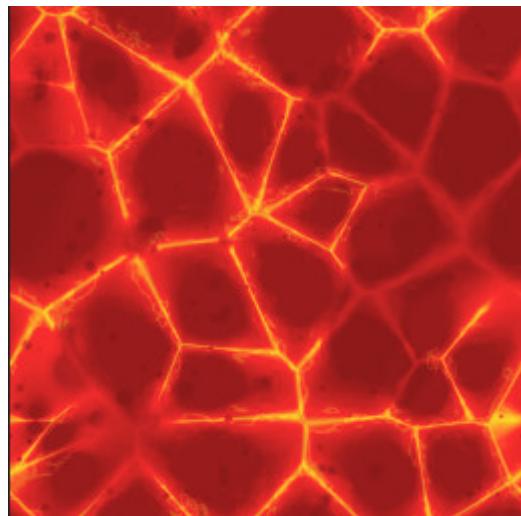
### 2.2.3 Zvuky a hudba

Většina her má soundtrack a zvukové efekty. Soundtrack obvykle nemá nijak zvlášť přísná pravidla, ale zvukové efekty musejí být výstižné a odpovídající akci v daný moment.

Jukebox [6] je model, který dokáže generovat hudbu se zpěvem v originální nezpracované formě zvukových dat s délkou v řádu minut, i s určením žánru a vokálního stylu. Modelů jako je tento již existuje více, avšak zatím to nejsou plně hodnotné soundtracky pro hry a ještě chvíli potrvá, než bude možné jednoduše vygenerovat hudbu a efekty pro hru pomocí pouhého nástroje.

#### 2.2.4 Textury

Jedná se o techniku vytváření textur pomocí algoritmů a matematických funkcí, namísto ručního malování, nebo použití statických obrázků. Tato metoda umožňuje tvůrcům vytvářet rozmanité textury s různými vlastnostmi a efekty, jako jsou organické vzory, terénní detaily, mraky nebo textury dřeva. Jednou z nejčastějších metod, která se používá na tvoření textur, jsou [L-systémy](#), nebo Perlinův šum, který je detailněji popsán v sekci [3.3](#).



Obrázek 2.4: Procedurálně vygenerovaná textura představující lávu.

## Kapitola 3

# Metody procedurálního generování

Tato kapitola popisuje metody pro generování geometrie, vegetace, celulární automaty, Perlinův šum a použití procedurálního generování v herním průmyslu, dále do detailu popisuje PG krajiny, generování fraktálů.

Procedurální modelování je téma, které se aktivně zkoumá už přes čtyřicet let. Myšlenka je, jak již bylo zmíněno, aby obsah který se vytvářel ručně, dal modelovat pomocí navržené procedury automaticky. Takovýto přístup se již uplatnil na generování například textur, geometrických modelů, zvukových nahrávek, nebo animací. V roce 1980 se začalo pracovat s různými metodami na vytváření terénu (hory, pláně a jezera). Začal se také řešit růst rostlin a obecně práce s přírodou. [41]

Roden and Parberry [40] pojmenovávají tento druh algoritmů *amplifikační algoritmy* (*amplification algorithms*), přijímají menší množství vstupních informací, které zpracují a vracejí větší objem dat na výstupu. Hendrikx et al. [15] pojímají procedurální generování jako alternativu k mechanickému navrhování obsahu, ale kladou důraz na zdokonalování a přidávání parametrů umožňujících zásah návrháře do takto vygenerovaných objektů.

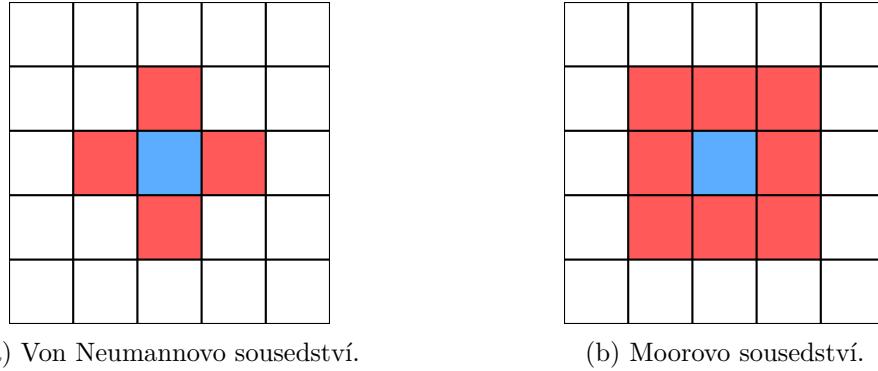
### 3.1 Celulární automaty pro procedurální generování

Celulární automaty se ve hrách používají intenzivně zejména pro modelování týkajícího se systémů v prostředí, jako jsou teplo, oheň, děšť, tlak a exploze. Zatím podle průzkumu nejsou známé žádné hry, které by postavily generování celého 2D herního světa, pouze pomocí celulárních automatů. Momentálně existují webové stránky, které generování malých map pomocí mřížek navrhují, ale není jich mnoho a neexistuje žádné spolehlivé ohodnocení těchto algoritmů. [17]

Původně byly CA vymyšleny Johnem Von Neumannem jako formální model sebereprodukujících se organismů. Šlo o dvou-dimenzionální celulární automat, kde každá buňka tzv. cell, je malý čtverec na velkém čtverečkovém papíru. Každá buňka má dva možné stavby černý a červený, které jsou určeny jejich sousedstvím. V John Von Neumannově teorii je sousedství tvořené čtyřimi přilehlými čtverci a na obrázku 3.1a jsou vyznačeny červenou barvou. [11]

Nejznámější celulární automat byl vytvořen v roce 1970 britským matematikem Johnem Hortonem Conwayem, který byl nazýván Game Of Life. Stejně jako Von Neumannův byl i tento automat dvou-dimenzionální a buňky mohly nabývat pouze hodnot živá, nebo mrtvá. Využívá Moorovo sousedství, které oproti Von Neumannově považuje za sousední buňky všech osm přilehlých, vyobrazené na obrázku 3.1b. Fungování automatu je následovné,

buňka zůstává naživu, pokud má dvě, nebo tři sousedící buňky živé. Což simuluje, že buňka nepřežije pokud je osamělá, ale zároveň pokud je okolí přeplněné organismy, tak je utlačována. Další pravidlo je, že pokud je libovolná buňka mrtvá, může se "narodit", pokud jsou v sousedství alespoň tři živé buňky. Toto pravidlo má simulovat rození, kde každá buňka musí mít tři rodiče. Automat díky těmto jednoduchým pravidlům dokáže vytvářet simulace které působily jako živý organismus. [11]



Obrázek 3.1: Sousedství buněk celulárních automatů z pohledu Von Neumanna a Moora.

## 3.2 L-systémy

L-systémy (Lindenmayerovy systémy) jsou formálním nástrojem [36], který se používá pro modelování vývoje rostlin (ukazuje obrázek 3.2) a buněčných struktur. Tyto systémy byly zavedeny biologem Aristidem Lindenmayerem v roce 1968. [12] Jedná se o paralelní řetězce přepisující systémy, za účelem modelovat růst celulárních organismů. L-systém  $\mathcal{L}$  je entice

$$\mathcal{L} = \langle M, \omega, R \rangle,$$

kde  $M$  je abeceda L-systému,  $\omega$  je axiom a  $R$  je množina pravidel přepisování. Abeceda obsahuje parametrizované moduly  $M = A(P), B(P), \dots$ , kde  $P = p_1, p_2, \dots, p_n$  jsou modulové parametry, jako jsou rotace, zvětšení, zmenšení. Axiom  $\omega \in M^+$  je neprázdná sekvence modulů a  $M^+$  jsou všechny možné prázdné řetězce z  $M$ . Pravidla pro přepisování mají následující formu:

$$id_1 : A(P) : cond \rightarrow x, x \in M^*,$$

$$id_1 : A(P) : cond \rightarrow x, x \in M^*,$$

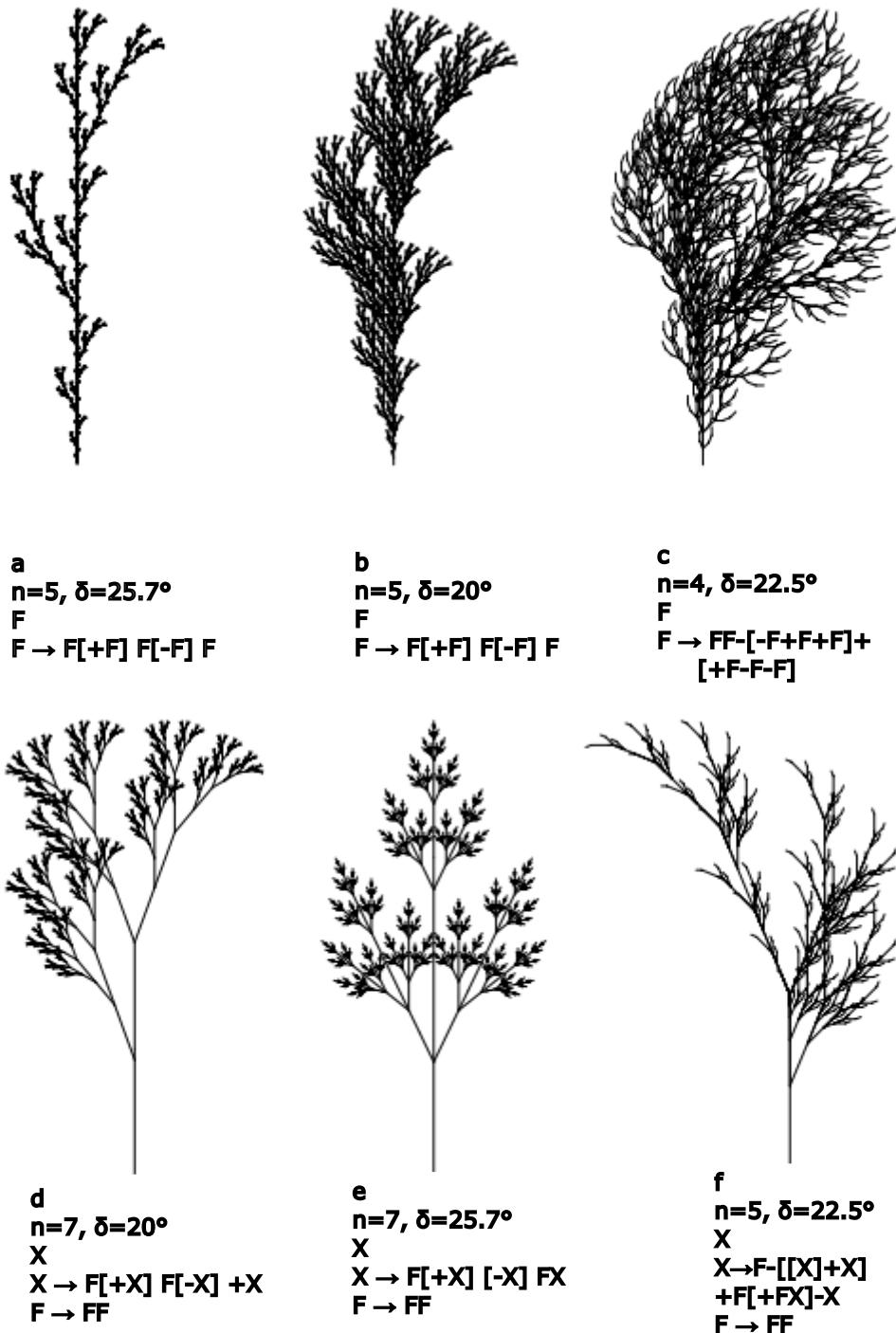
...

kde  $M^*$  jsou všechny možné řetězce z  $M$  včetně prázdného  $\epsilon$ . Pravidlo  $id_i$  přepisuje znak na levé straně z abecedy  $A(P)$  posloupností písmen z pravé strany, pokud je podmínka  $cond$  pravdivá. Modul, který se nenachází na levé straně pravidla, se nazývá *terminální symbol*, neboť se nemůže dál měnit, všechny ostatní moduly se nazývají *neterminální symboly*. [39]

Každé písmeno má vlastní pravidlo derivace řetězce, ale probíhá paralelním provedením aplikovatelných pravidel, z množiny  $R$  pro každé písmeno které obsahuje. Produkční pravidla přepisují začínající symbol sekvencí modulů a pokračují v úspěšných derivacích  $\omega \Rightarrow m_1 \Rightarrow m_2 \Rightarrow \dots$ , dokud není možné žádný další modul přepsat (řetězec končí pouze

terminálními symboly), řetězec modulů je prázdný, kvůli aplikaci pravidla epsilon, nebo byl proces ukončen kvůli maximálnímu počtu iterací, které stanovil uživatel.[25]

Rekurze nastává v L-systémech tehdy, když se symbol z levé strany objevuje na pravé straně toho stejného pravidla (i když ne přímo). Nedeterminismus povoluje více pravidel pro jeden znak z abecedy. Toto navíc vyžaduje specifikaci pravděpodobnosti jejich aplikace.[26]

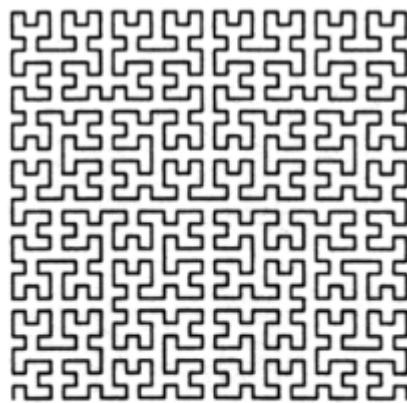


Obrázek 3.2: L-systém generující fraktály řídící se pravidly, které připomínají rostliny.

### 3.2.1 Geometrie pomocí L-systémů

Pro tvoření geometrie z textových řetězců, je každý řetězec reprezentován želvou, která vytváří geometrické symboly, jako jsou čáry, nebo dokonce 3D geometrii. Ve 2D má želva stav  $S(p, 0)$  kde  $p = [x, y]$  je její pozice a 0 je směrový vektor, který udává směr jejího pohybu.

Želva sekvenčně čte písmena interpretovaného řetězce z modulu od začátku po konec, kde každé písmeno je interpretované jako příkaz. Písmeno F si překládá jako "pohyb od  $p$  směrem k 0 o vzdálenosti d, která je zadaná a nakreslí linku mezi starou pozicí a novou." Příkazy  $+(\alpha)$  a  $-(\alpha)$  mění směr pohybu želvy, jejím otočením doleva, či doprava o  $\alpha$ . Cokoliv v závorkách  $[M^+]$  je geometricky interpretováno, jako větev vygenerované struktury. Výhodou je, že ne všechna písmena abecedy musí mít nastavenou geometrickou interpretaci, pokud ji nemají, tak je želva ignoruje. [36]



n=5, d=6  
X  
X → -YF+XF+FX+FY-  
Y → +XF-YFY-FX+  
F → F    + → +

Obrázek 3.3: Vygenerovaných obrázků L-systémy želví interpretací, jsou zde vidět jednotlivé posuvy i hodnoty, o které se posouvá.

## 3.3 Šumy

Šumy se v počítačové grafice využívají například pro přidání kvalitních detailů do synteticky vytvořených obrázků. Perlinův šum, navržený Kenem Perlinem [33], se v dnešní době používá ve vytváření procedurálních textur včetně mraků, vln, tornád, raketových cest, atd. Tato kapitola poskytuje detailní přehled jednotlivých funkcí generujících šum. Rozděluje tyto funkce do tří kategorií: gradientní mřížkové šumy 3.3.2, explicitní šumy 3.3.3 a řídké konvoluční šumy 3.3.4. V každé z těchto kapitol je popsáno několik reprezentativních šumových funkcí (rozebraných do detailu) a další související příklady [19].

### 3.3.1 Definice šumu

Šum je generátor náhodných čísel počítačové grafiky. [33] Jedná se o náhodný a neuspořádaný vzor, který je užitečný všude tam, kde je potřebný detail bez evidentní struktury. Jednoduchý šum pracuje takto:

1. Uvažujme množinu všech bodů v prostoru, jejichž souřadnice  $x, y, z$  jsou všechny celočíselné. Tuto množinu nazveme celočíselná mřížka. Každému bodu v této mřížce je přiřazena pseudonáhodná hodnota a gradientní hodnoty  $x, y$  a  $z$ . Přesněji, zobrazte každou uspořádanou posloupnost tří celých čísel do nekorelované uspořádané posloupnosti čtyř reálných čísel:  $[a, b, c, d] = H([x, y, z])$ , kde  $[a, b, c, d]$  definují lineární rovnici s gradientem  $[a, b, c]$  a hodnotou  $d$  v bodě  $[x, y, z]$ .  $H$  je nejlépe implementováno jako hashovací funkce.
2. Pokud  $[x, y, z]$  je na celočíselné mřížce, definujeme  $Noise([x, y, z]) = d_{[x, y, z]}$ . Pokud  $[x, y, z]$  není v celočíselné mřížce, spočítáme hladkou (např. kubickou polynomickou) interpolaci mezi koeficienty rovnic mřížky, aplikovanou nejprve v  $x$  (podél hran mřížky), pak v  $y$  (ve vnitřních plochách mřížky  $z$ ) a nakonec v  $z$ . Poté vyhodnotíme tuto interpolovanou rovnici v bodě  $[x, y, z]$ .

Ohodnocením hodnot takového šumu jsme schopni vytvářet jednoduché náhodné textury povrchu. [33]

Například při ohodnocení šumu pouze bílou barvou:

$$color = white * Noise(point)$$

Výše uvedená textura má omezený frekvenční charakter, není zde žádný detail mimo určitý rozsah velikosti. Vzniklý obrázek je 3.4a.

S hodnotou vrácenou funkcí  $Noise()$  [34] lze dále dělat mnoho různých věcí, s pomocí funkční kompozice, lze například mapovat různé rozsahy hodnot do různých barev:

$$color = Colorful(Noise(k * point))$$

V příkladu výše, byla textura škálována pomocí násobení domény funkce  $Noise()$  konstantou  $k$ . Jednou z výhod přístupu funkční kompozice je jednoduchost, se kterou lze takovéto úpravy provádět. Výsledný obrázek je níže 3.4b.



(a) Torus s bodovým vzorem.

(b) Torus s barevným vzorem.

Obrázek 3.4: Na obrázcích jsou vidět tory se vzory podle aplikovaných ohodnocení šumů, obrázky jsou převzaty z [33].

### 3.3.2 Gradientní mřížkové šumy

*Gradientní mřížkové šumy* vytvářejí šum pomocí interpolace, nebo konvolucí náhodných hodnot, nebo gradientů definovaných v bodech celočíselné mřížky. Reprezentativním příkladem je **Perlinův šum** [19].

#### Perlinův šum

Roku 1985 Perlin představil *Perlinův šum*, jeho slavnou procedurální funkci pro generování šumu. [33, 34] Perlinův šum určuje šum v bodě prostoru výpočtem pseudonáhodného gradientu u každého z osmi nejbližších vrcholů na celočíselné krychlové mřížce a následným provedením spline interpolace. Pseudonáhodný gradient je získán hashováním mřížkového bodu a použitím výsledku k výběru gradientu. Mřížkové body jsou hashovány pomocí postupné aplikace pseudonáhodné permutace na souřadnice k dekorelaci indexů do pole pseudonáhodných jednotkových gradientových vektorů. Sada gradientů se skládá z 12 vektorů definovaných směry od středu krychle k jejím hranám. Interpolantem je kvintický polynom, který zajišťuje spojitou derivaci šumu, neboli hladké přechody mezi jednotlivými body.

Již od svého uvedení před skoro čtyřmi desetiletími se Perlinův šum široce využívá v grafice, například při generování textur, které je vidět na obrázku 3.5. Perlinův šum je rychlý, jednoduchý a nadále zůstává pilířem průmyslu.



Obrázek 3.5: Váza s vygenerovanou texturou pomocí Perlinova šumu. Obrázek převzat z [33].

### Lepší gradientový šum

Modifikovaná hashovací funkce kombinovaná s oddělenou gradientní tabulkou vylepšuje axiální dekorelaci. Jiné rekonstrukční jádro zlepšuje omezení pásmu. Metoda projekce zlepšuje kvalitu šumu na 2D površích pomocí pevného šumu. Je třeba poznamenat, že tyto zlepšení platí pro několik druhů gradientových šumů mřížky. [18]

### Hardwareové implementace

Již mnoho autorů představilo hardwareové implementace funkcí podobných Perlinovu šumu. Hart a spol. [14] prezentovali VLSI hardwareovou implementaci Perlinova šumu. Dále byly prezentovány GPU implementace Perlinova šumu, jak od Harta [13], tak i od Olano [32]. Dále je Perlinův šum nedílnou součástí OpenGL Shading Language (GLSL).

#### 3.3.3 Explicitní šumy

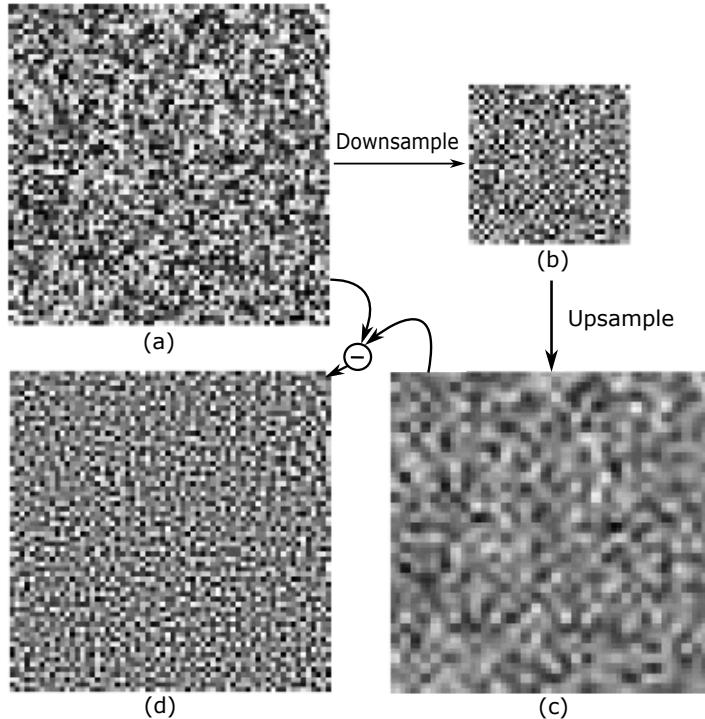
*Explicitní šumy* generují šum explicitním způsobem (předzpracováním) a ukládají ho. Do slovně vzato, explicitní šumy nejsou procedurální šumové funkce, ale i tak jsou velmi podstatné. Dva reprezentativní příklady jsou: **vlnkový šum** a **anisotropní šum** [19].

### Vlnkový šum

Poprvé představen autory Cook a DeRose roku 2005 [5]. Perlinův šum ve větších vzdálenostech mívá problém s aliasingem a ztrátou detailů, z důvodu slabého pásmového omezení, kvůli tomu byla představena nová šumová funkce, která je téměř perfektně pásmově omezená.

Podstata algoritmu spočívá v následujících čtyřech krocích, které jsou vyobrazeny na obrázku 3.6:

1. Vytvoření obrazu  $R$  vyplněného náhodným šumem.
2. Snížení vzorkovací frekvence  $R$  za účelem vytvoření obrazu o poloviční velikosti  $R^{\downarrow}$ .
3. Zvětšení  $R^{\downarrow}$  na plnou velikost  $R^{\downarrow\uparrow}$ .
4. Odečtení  $R^{\downarrow\uparrow}$  od originálního obrazu  $R$  aby byl vytvořen výsledný obraz  $N$ .



Obrázek 3.6: Proces generování šumu.(a) Obrázek  $R$  náhodného šumu, (b) o polovinu menší obrázek  $R^{\downarrow}$ , (c) poloviční rozlišení obrázku  $R^{\downarrow\uparrow}$ , (d) obraz pásmového šumu  $N = R - R^{\downarrow\uparrow}$ . Obrázek převzat z [5]

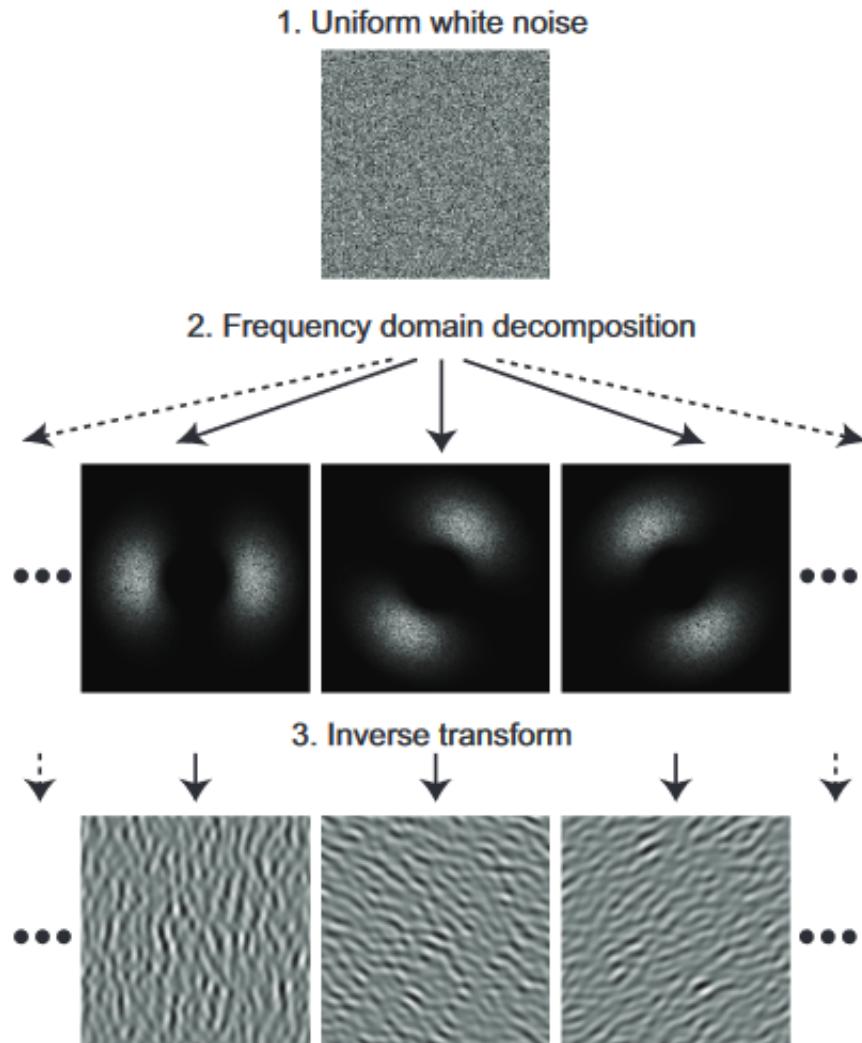
Koefficienty šumu v dlaždici  $N$  jsou tedy vytvořeny z  $R$  a odstraněním části, která je reprezentovatelná na poloviční velikosti. Zbývá část, která není reprezentovatelná na poloviční velikosti, tj. pásmově omezená část. Filtry použité v krocích vzorkování dolů a nahoru jsou získány pomocí vlnkové analýzy. Rozšíření na více dimenzí je přímé.

### Anisotropní šum

Roku 2008, představil Goldberg a spol. *anisotropní šum* [10]. Goldberg vypozoroval, že existující šumové funkce podporují pouze isotropní filtrování, na což se také vztahuje aliasing a ztráta detailů a představil novou funkci, která podporuje anisotropní filtrování na vysoké úrovni.

Hlavní myšlenkou bylo vygenerovat šumové textury tak, že frekvenční obor je rozdělen do orientovaných pod-pásů. Anisotropní šumová pásma mají nejen určitý rozsah škály, ve kterém jsou účinné, ale také mají preferovanou směrovou orientaci. Konstrukce takového šumu je založena na ovladatelných filtroch, které rozdělují frekvenční obor. Poskytuje řadu vlastností, které jsou zásadní pro generování šumu:

1. Každý filtr definuje pod-pásмо, které je úzce lokalizované velikostí a orientací.
2. Filtry implementují invertibilní transformaci. To znamená, že lze přesně obnovit signál z jeho dekompozice do pod-pásem.
3. Filtry disponují možností točení orientace. Hlavním účelem je, že lineární interpolace filtrování může generovat filtr s přesně stejným profilem, ale s prostřední orientací.



Obrázek 3.7: Ilustrace spektrálního generování šumu. Dekompozice frekvenční domény má tři orientace. Jsou vidět tři orientované pod-pásma se stejnou velikostí a odpovídajícími obrazy prostorových domén, které se následně ukládají jako textury. Obrázek převzat z [10].

Každý orientovaný pod-pásmový obraz je zabalen do jednoho kanálu 32bitového RGBA obrazu, což vede ke čtyřem orientacím na každou texturu. Obvykle použití čtyř nebo osmi pásů, tj. jedné nebo dvou textur, představuje dobrý kompromis mezi úložným prostorem, rychlostí vykreslování a kvalitou obrazu. Pod-pásma šumu jsou vypočítána dopředu pouze na jedné škále najednou. Všechny ostatní škály jsou generovány za běhu jednoduše pomocí stupňování předem vypočítaných textur. [19]

## Stochastické poddělení

Fournier a spol. [7], který představil metodu středního posunu, uvedl také stochastický algoritmus dělení pro generování přírodních nepravidelných fraktálních objektů a jevů, jako je terén. Lewis [21, 22] předvedl zobecněné stochastické poddělení a generalizoval Fournierovu práci na libovolné autokorelační funkce.

## Fourierova spektrální syntéza

Generuje šum s konkrétním výkonovým spektrem filtrováním bílého šumu ve frekvenční oblasti. Fourierova spektrální syntéza byla uvedena v počítačové grafice autorem Anjyo [2], Saupe a Voss [3], kteří ji využili, aby vygenerovali náhodné fraktály a simulovaly přírodní jevy. Fourierova spektrální syntéza může být užitečná i při generování referenčních řešení pro šumové funkce, u kterých je známé očekávané výkonové spektrum.

### 3.3.4 Řídké konvoluční šumy

Generují šum jako sumu náhodně pozicovaných a vážených kernelů. Třemi reprezentativními příklady jsou: šum řídké konvoluce, **bodový šum** a Gaborův šum.

#### Definice

V sérii prací mezi lety 1984 a 1989 Lewis představil *řídký konvoluční šum* [24, 21, 23].

Konstrukce řídkého konvolučního šumu je jednoduchá: libovolné jádro  $k$  je konvoluováno se šumem Poissonova procesu  $\gamma$ .

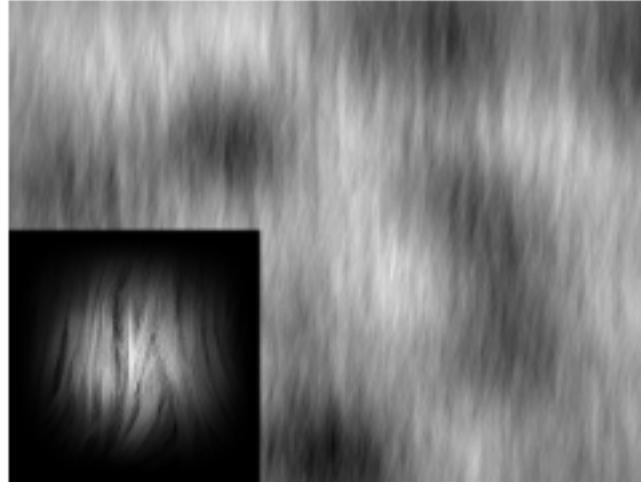
$$N(x, y) = \int \gamma(u, v)k(x - u, y - v)dudv$$

Poissonův proces sestává z impulsů nekorelovaných intenzit  $a_k$  situovaných na náhodných nezávisle na sobě zvolených pozicích  $(x_k, y_k)$ .

$$\gamma(x, y) = \sum_k a_k \delta(x - x_k, y - y_k)$$

Poissonův proces je řídký, což znamená, že nedefinuje každý pixel nebo bod v prostoru, od toho je odvozený název *řídká konvoluce*. Tento způsob použití řídkého impulsního šumu umožňuje alespoň nějakou výpočetní efektivitu, vzhledem k tomu, že konvoluce efektivně rozprostírá jádro se škálou amplitudy pouze na určitých místech  $(x_k, y_k)$ .

Abychom vyhodnotili šum v konkrétním bodě, je nutné použít pouze jádra, která se s tímto bodem překrývají. Toto je urychleno zavedením virtuální mřížky, kde velikost buňky mřížky odpovídá poloměru jádra. Vyhodnocení pak zahrnuje pouze jádra, která jsou umístěna v buňce obsahující daný bod a těch v sousedních buňkách. Souřadnice buňky jsou také použity k zasazení generátoru náhodných čísel pro generování Poissonových impulzů umístěných v této buňce. Více podrobností a vylepšených schémat pro tento krok jsou uvedeny u Worleyho [45] a Lagae a spol. [20]. I když Lewis [23] popisuje několik optimalizací, jako je ukládání konstruovaných Poissonových impulzů za předpokladu koherentního přístupu, je řídká konvoluční metoda trochu pomalejší než jedna oktáva Perlinova šumu. Jelikož výkonové spektrum výstupu konvoluce je součinem vstupů a výkonové spektrum Poissonova impulsu je konstantní, je jednoduše škálovánou verzí tohoto jádra.

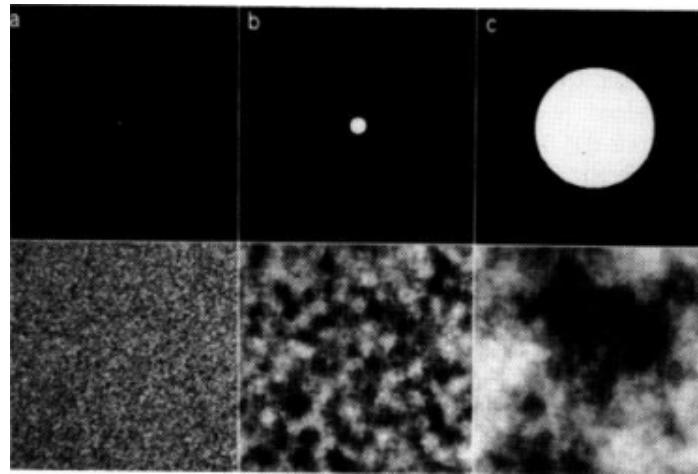


Obrázek 3.8: *Rídka konvoluční šum*. Vlevo dole: Vzorek obrázku vlasů. Vpravo: přibližná textura vlasů vytvořená pomocí 2D řídké konvoluce. Obrázek převzat z [19].

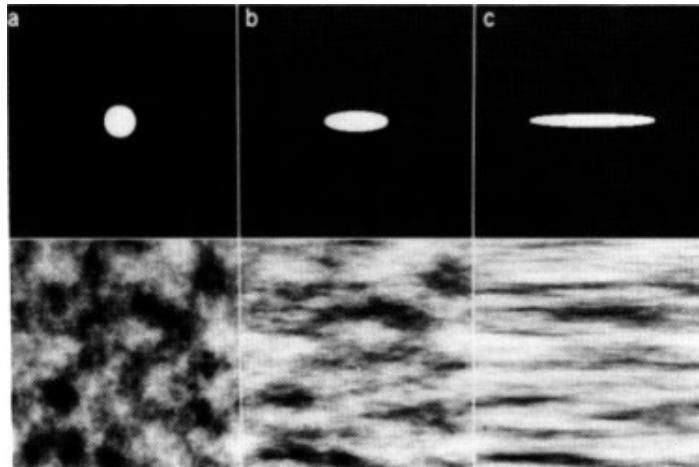
### Bodový šum

Jedná se o metodu generování stochastických textur pro vizualizaci skalárních a vektorových polí nad plochami. Na bodový šum lze nahlížet jako na explicitní formu řídkého konvolučního šumu vypočítaného konverzí bodů, nebo Fourierovou spektrální syntézou (popisána v sekci 3.3.3).

Van Wijk [44] diskutuje vztah mezi bodem a texturou v detailech. Poukázal na několik důležitých konceptů, které později uvedl v kontextu šumu například: mapování textur na parametrických plochách, syntézu textur nad zakřivenými plochami jako alternativu pevnému šumu, a lokální kontrolu variací bodu. Dále lze pracovat s tvary, ostrostí hran, nebo různými vzory.



Obrázek 3.9: Vztah mezi texturou a bodem. Na třech dvojicích obrázků jsou vidět tři různé velikosti bodu a tři různé šumy které se k těmto bodům vztahují. Obrázek převzat z [44].



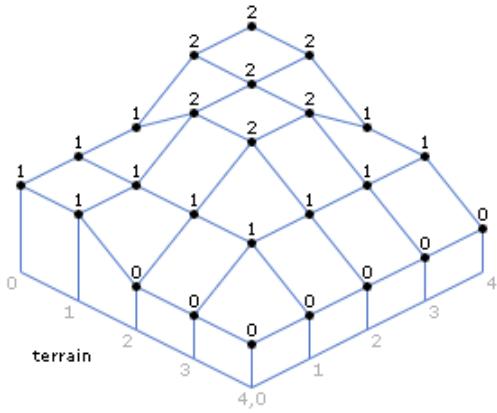
Obrázek 3.10: Na třech dvojicích obrázků je vidět jak se textury mění, když se velikost obrázku mění ne-proporcionalně. Obrázek převzat z [44].

### 3.4 Výškové mapy

Výškové mapy představují dvoudimenzionální mřížky obsahující výškové hodnoty, jež jsou častým prvkem v modelování terénu. Tyto mapy se běžně využívají jako klíčový prvek pro reprezentaci základu terénu v herním průmyslu. Pro tvorbu výškových map existuje mnoho algoritmů. Na obrázku 3.11 je jednoduchá ukázka, jak jednotlivé body na mapě mají hodnoty, podle kterých se dále určí jejich elevace. [16]

Jedny z nejstarších algoritmů jsou metody založené na pododdelení. Segmenty v rámci vygenerované hrubé výškové mapy jsou iterativně rozdělovány, kde každá iterace navíc používá kontrolovanou náhodnost k přidávání detailů. Miller [28] popisuje některé varianty všeobecně známé metody středového posunu, ve které se výška nového bodu nastavuje na průměr hodnot jeho rohů v trojúhelníkovém nebo diamantovém tvaru, k němuž je přidán náhodný offset. Každou iterací se rozsah náhodných hodnot offsetu snižuje podle parametru, který kontroluje hrubost výsledné výškové mapy.

Generování výškových map se v dnešní době provádí převážně pomocí fraktálních generátorů šumu, jako je **Perlinův šum**, který provádí generování šumu vzorkováním a interpolací bodů na mřížce náhodných vektorů. Výškové mapy se mohou dále transformovat na základě běžných filtrací obrazu, například vyhlazování (smoothing), nebo simulací fyzikálních jevů jako je eroze. [41]



Obrázek 3.11: Ukázka hodnot a jejich výšek výškových map.

Jednou z hlavních nevýhod výškových map je, že nepodporují tvorbu skalních převisů a jeskyní. Gamito a Musgrave [9] navrhovali systém deformace terénu, který má za výsledek pravidelné, uměle vytvořené skalní převisy. O něco novější metoda [35] přináší propracovanější struktury s rozdílnými vrstvami materiálů, které podporují kameny, klenby, převisy a jeskyně. Příklady takto vygenerovaných struktur jsou vidět na obrázcích. 3.12



(a) Vygenerované převisy.

(b) Vygenerované kamenné klenby.

Obrázek 3.12: Na obrázcích jsou vidět klenby a převisy vygenerované pomocí algoritmu vytvořeného Peytaviem, obrázky jsou převzaty z [35]

# Kapitola 4

## Návrh řešení

Tato část se věnuje představení klíčových technologií využitých při vývoji hry a zdůvodnění toho, proč byly zvoleny. Dále se zabývá metodami, jak budou procedurální mapy vytvářeny, a představuje návrhy algoritmů a postupů pro generování oblastí.

### 4.1 Vybrané technologie

V mé práci jsem se rozhodl pro použití herního enginu Unity, který je podrobněji popsán v sekci 1.3. Před samotným začátkem práce jsem zhodnotil, že vývojové prostředí Unity nabízí mnoho výhod a pro vývoj procedurálního generování a následné hry bude ideální.

Dalším z důvodů pro volbu Unity je jazyk C#, který nabízí mnoho knihoven, které budou užitečné při tvorbě hry. Výhodou je, že C# disponuje silnou typovou kontrolou, což znamená, že chyby v kódu jsou odhaleny během překladu, což usnadňuje odhalování a opravování chyb při vývoji.

C# nabízí širokou škálu knihoven, ale tou pro mě nejdůležitější byla knihovna Mathf, která kromě mnoha matematických funkcí obsahuje také funkci Perlinova šumu. Jazyk C# je obecně velmi užívaný, a tím pádem má i velkou aktivní komunitu vývojářů přispívajících do knihoven, nástrojů, frameworků, poskytují podporu a návody, které pomáhají ostatním vývojářům efektivně pracovat s procedurálním generováním.

Některé další výhody použití programovacího jazyka C# v kombinaci s Unity pro tvorbu her s procedurálním generováním zahrnují:

**Snadná integrace s Unity:** C# je primárním programovacím jazykem pro vývoj her v Unity, což znamená že má těsnou integraci s Unity API a prostředím.

**Objektově orientovaný přístup:** C# je objektově orientovaný jazyk, což umožňuje vytvářet modulární a znovupoužitelný kód. To usnadňuje organizaci a správu algoritmů a umožňuje snadnou rozšířitelnost, údržbu a znovupoužitelnost v jiných projektech.

**Platformní nezávislost:** Díky tomu, že Unity podporuje mnoho různých platform, může být C# kód psaný v Unity použit na vytváření her pro různé platformy, včetně mobilních zařízení, stolních počítačů, konzolí a webu, což zvyšuje dostupnost a dosah hry.

Vývojovým prostředím jsem zvolil Visual Studio, které je primárním IDE doporučované společností Unity pro vývoj v jazyce C#. Poskytuje silnou integraci s Unity Editorem, což

znamená, že lze snadno vytvářet a upravovat skripty přímo v rámci Unity Editoru. Existuje ale několik dalších důvodů, proč zvolit Visual Studio jako vývojové prostředí:

**Široká podpora a komunita:** Visual Studio je velmi populární vývojové prostředí s rozsáhlou komunitou uživatelů a dostupností online dokumentace a návodů. To je užitečné pro získání podpory a řešení problémů během vývoje.

**Pokročilé funkce pro vývoj:** Nabízí mnoho pokročilých funkcí pro vývoj v jazyce C#, jako je inteligentní vyplňování kódu, refactoring, ladění za běhu a integrace s verzovacími systémy.

**Podpora pro rozšíření:** Visual Studio umožňuje rozšiřování funkcí pomocí různých rozšíření (balíčků a doplňků), která mohou usnadnit proces vývoje a zvýšit produktivitu.

## 4.2 Návrh procedurálního generování oblastí

V této sekci je popsán způsob jakým se generují jednotlivé oblasti, použité algoritmy, metody a různé úpravy, pomocí kterých bude vytvořena 2D mapa, na které se následně bude odehrávat samotná hra, jejíž návrh je detailněji popsán v sekci 4.3.

### 4.2.1 Zvolený šum

Pro generování samotného šumu jsem vybral známý Perlinův šum (který je detailně popsán v sekci 3.3.2), který je relativně jednoduchý na implementaci a jeho porozumění. Algoritmus, kterým je implementován, je intuitivní a snadno přenositelný do různých herních prostředí. Zde je několik dalších důvodů, proč byl vybrán právě Perlinův šum pro generování:

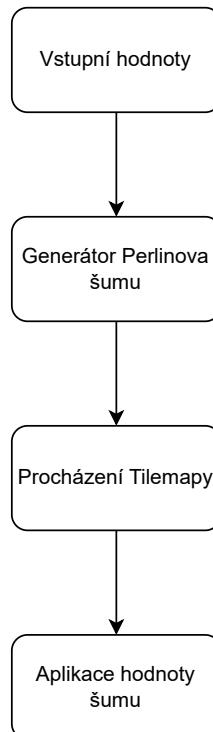
**Hladkost a přirozenost:** Perlinův šum poskytuje hladké a přirozené výsledky. Jeho charakteristika umožňuje vytvářet organické a realistické terény.

**Efektivita:** Perlinův šum je poměrně efektivní z hlediska výpočetní náročnosti. Jeho algoritmus umožňuje generování šumu s rozumným výkonem a nízkou paměťovou náročností.

Oproti například vlnkovému šumu, sekce 3.3.3, sice nemá takovou flexibilitu v nastavení detailů a škálování, ale kvůli přirozenosti jsem se rozhodl pro Perlinův. Perlinův šum bude mít mnoho vlastností (které jsou detailně popsány v sekci implementace ??), ale asi nejdůležitější bude vlastnost *seed*, pomocí které bude možné kteroukoliv mapu znova vygenerovat.

#### 4.2.2 Algoritmus generování mapy pomocí Perlinova šumu

Procedurální generování mapy je jedním z hlavních cílů této práce. Pro generování samotné mapy se bude využívat již zmíněný Perlinův šum s použitím výškových map, které jsou popsány v sekci 3.4. Každá oblast, země, voda, skály, nebo pláže, bude mít definovaný rozsah výšky na výškové mapě. Unity disponuje nástrojem zvaným *tilemap*, jedná se o čtvercovou mřížku, do jejíž buněk lze skládat takzvané *tiles* (dlaždice). Velkou výhodou těchto tilemap je, že každý bod reprezentuje jeden tile, tím pádem po ohodnocení bodu lze pouze položit odpovídající tile.



Obrázek 4.1: Ilustrace navrhovaného diagramu algoritmu, který bude generovat mapu.

Kvůli jednodušší a srozumitelné interakci s uživatelem při generování map v Unity, je navržen skript, obsahující následující algoritmus (který je vyobrazený na obrázku 4.1):

**Vstupní hodnoty:** Uživatel zadá potřebné hodnoty pro generování šumu, jako jsou výška a šířka mapy a přiblížení šumu. Tyto hodnoty určují vlastnosti výsledného šumu a tím i vzhled herního terénu.

**Generátor Perlinova šumu:** Na základě zadaných parametrů se generuje dvoudimenzi-  
onální Perlinův šum. Tento šum je vytvořen pomocí algoritmu Perlinova šumu, který produkuje hladké a přirozeně vypadající přechody mezi různými hodnotami.

**Procházení tilemapy:** Tilemapa je komponenta, která vytváří mřížkovanou plochu, na jejíž buňky lze následně nanášet sprity. Procházením každého bodu na tilemapě se určí, jaký typ terénu nebo objektu, by měl být umístěn na daném místě. To se provádí porovnáním hodnoty Perlinova šumu v daném bodě s definovanými prahovými hodnotami nebo regiony.

**Applikace hodnoty šumu:** Na základě porovnání s definovaným regiony se do každého bodu na tilemapě vloží odpovídající tile, který reprezentuje určitý typ terénu, jako jsou travnaté plochy, hory, jezera nebo lesy. Díky tomu, může být vytvořen pestrobarevný a realistický herní svět.

Tento algoritmus poskytuje uživatelům možnost tvorit rozmanitá a realistická herní prostředí v Unity pomocí Perlinova šumu a tilemap. Samotný algoritmus je navržen tak, aby poskytoval, co největší kontrolu na vzhledem vygenerovaných map. Díky flexibilitě a jednoduchosti tohoto přístupu může být proces tvorby herního světa intuitivní a rychlý.

#### 4.2.3 Algoritmus Wave Function Collapse

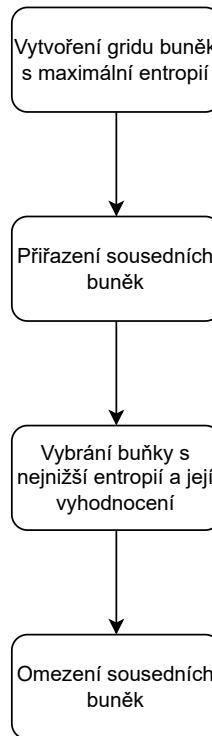
Při návrhu této práce jsem se rozhodl zkoumat různé metody procedurálního generování terénu. Mezi ně patří již zmíněný Perlinův šum, který již má dlouhou historii v počítačové grafice. Nicméně se tato práce zaměřuje i na novější přístupy generování terénu, jako je *Wave Function Collapse* (WFC), který nabízí zcela odlišný přístup k procedurálnímu generování.

#### Důvody pro volbu Wave Function Collapse

Wave Function Collapse byla vybrána pro svou významnou flexibilitu a rozmanitost při generování struktur s komplexními vzory a detaily, které jsou obtížně dosažitelné pomocí konvenčních metod. Její deterministická povaha zajišťuje konzistentní a reprodukovatelné výstupy, což je v oblasti procedurálního generování významným přínosem. Díky tomu lze dosáhnout zaručené kvality generovaného obsahu, což je klíčové pro mnohé aplikace. Tato metoda je navíc modulární a lze ji aplikovat na širokou škálu problémů, což ji činí atraktivním nástrojem pro různé oblasti aplikací.

#### Princip Wave Function Collapse

Wave Function Collapse pracuje s prostorovým modelem dat, který je reprezentován jako mřížka buněk. Každá buňka může nabývat různých stavů z definovaného souboru možností. Wave Function Collapse iterativně vyplňuje prostor stavů tak, že se snaží dodržovat lokální konzistence mezi sousedními buňkami. To znamená, že vyplněný prostor musí splňovat určité podmínky nebo omezení definované vstupními pravidly. Algoritmus postupně prochází prostor stavů a aplikuje pravidla lokální konzistence, která určují, jakých stavů mohou sousední buňky nabývat na základě již vyplněných buněk. Pokud algoritmus není schopen vyplnit prostor stavů podle vstupních pravidel, může dojít k zpětnému kroku a změně vyplněných buněk. Proces iterace pokračuje, dokud existuje ještě nevyplněná buňka.



Obrázek 4.2: Ilustrace navrhovaného algoritmu Wave Function Collapse, který bude generovat mapu.

### 4.3 Návrh hry

Tato sekce popisuje žánr, styl, cíle a ovládání samotné hry, založené na procedurálním generování 2D map v herním prostředí. Hra bude kombinovat prvky real-time strategie (RTS) s prvky survival žánru, vyžaduje strategické plánování a rychlé rozhodování v rámci nepřetržitého boje o přežití. Cílem hry je bránit se pravidelným nájezdům nepřátel, zatímco hráč současně buduje a rozvíjí svoji obrannou sílu.

Při výběru hratelné rasy a dvou začínajících postav na začátku hry hráči rozhodují o strategickém směru, kterým se jejich osada bude ubírat. Různé rasy mohou poskytovat specifické výhody a jedinečné schopnosti, což ovlivňuje strategii budování a obrany hráčů. Kromě budování základní infrastruktury, jako jsou farmy pro zajištění potravin, musí hráči také efektivně využívat dostupné prostředky a suroviny k budování různých typů obranných struktur. To může zahrnovat stavbu opevněných hradeb, věží, pastí, které mají za cíl zpomalit, nebo zastavit postup nepřátelských sil.

Kromě obrany před nepřáteli se hráči budou muset vypořádat s dalšími přírodními a lidskými hrozbami, jako je nedostatek surovin, nebo náhlé změny počasí. Vývoj ve hře je doprovázen možností rozšíření a vylepšení stávajících budov, což umožňuje vybudovat silnější a odolnější obranné systémy. Zároveň se hráči musí adaptovat na nové výzvy a nepředvídatelné situace.

#### 4.3.1 Grafika

Hra je navržena jako 2D hra s pohledem zhora (top-down), jako je vyobrazeno na obrázku 4.3, což poskytne hráčům přehledný pohled na herní prostředí a umožní jim snadnou navigaci a řízení svých jednotek. Grafický styl hry bude pixelový s použitím převážně 16x16 pixelových spriteů. Tento styl grafiky je známý svou jednoduchostí. Modelů tohoto stylu je mnoho volně dostupných a v případě potřeby lze bez problémů dotvořit vlastní. Díky tomuto stylu grafiky bude zároveň možné udržet nízké nároky na hardwarové prostředky, což umožní hladký běh hry i na starším zařízení.



Obrázek 4.3: Ukázka grafického stylu hry s postavami hráčů a stromů. V levém dolním rohu je vidět stavební menu a v horním rohu panel se surovinami.

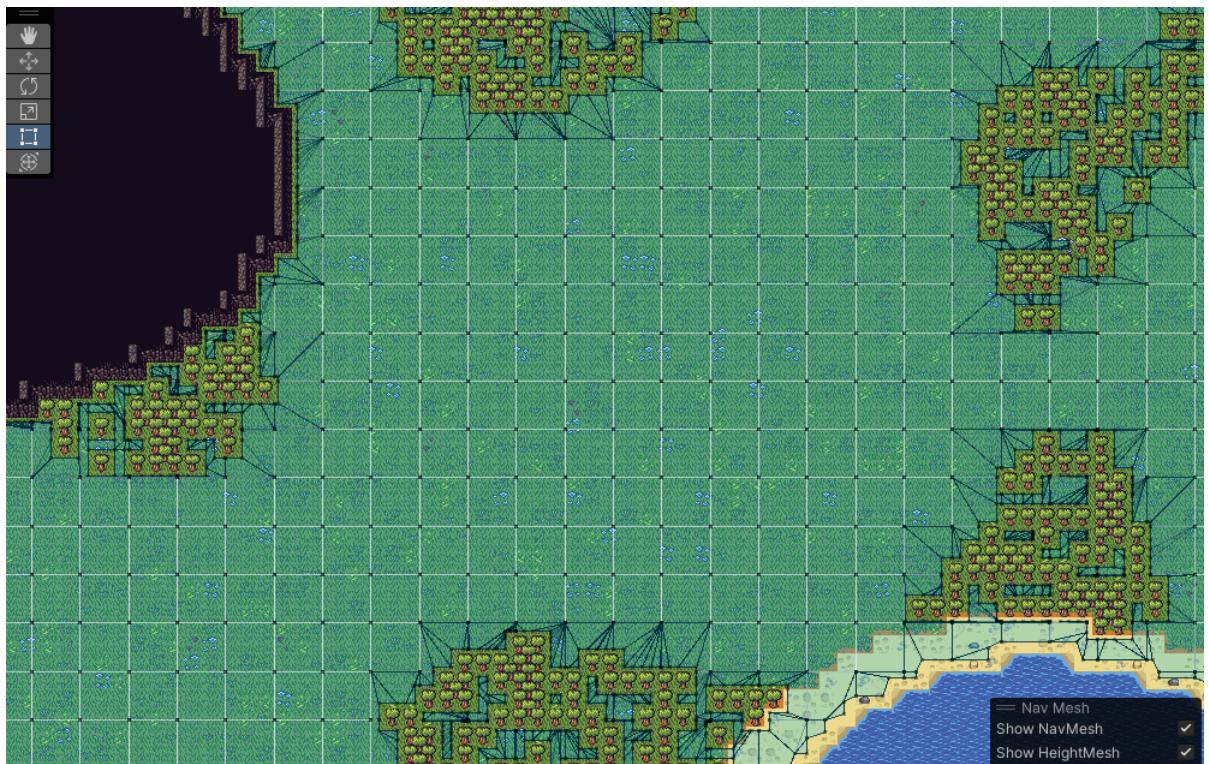
#### 4.3.2 Jednoduchá umělá inteligence nepřátel

Umělá inteligence nepřátel bude implementována pomocí skriptů, které budou mít za úkol detektovat hráče v okolí a podle toho reagovat, včetně pronásledování a útoku na hráče. Pro pohyb po mapě bude využit vestavěný systém v Unity nazývaný NavMesh. Každá jednotka, která se má pohybovat po mapě, bude mít přiděleného NavMesh agenta, který bude schopen reagovat na podloží definované NavMesh surface. Tímto způsobem bude umožněno nepřátelským jednotkám plánovat svůj pohyb po herním světě s ohledem na dostupné průchody a neprůchodné oblasti definované v NavMesh surface, což vytvoří realističtější a dynamické chování nepřátel.

#### 4.3.3 NavMesh funkcionalita

NavMesh je vestavěný systém v Unity, který umožňuje jednotkám v herním světě plánovat svůj pohyb a navigovat přes prostředí. Jedná se o komplexní nástroj pro tvorbu navigačních dat, která definují, kde se mohou jednotky pohybovat a kde nemohou. Na obrázku 4.4 je vidět připravená navigační síť NavMesh. NavMesh systém v Unity funguje následovně:

1. Nejprve je třeba vytvořit navigační síť v herním světě. To se obvykle provádí pomocí generování NavMesh surface, které automaticky vytvoří navigační data podle geometrie herního světa.
2. Poté co je NavMesh vytvořen, mohou být jednotkám přiděleni NavMesh agenti. Jedná se o komponentu umožňující jednotkám navigaci přes NavMesh a plánovat jejich pohyb.
3. Agenti mohou plánovat svůj pohyb v reálném čase na základě polohy jednotky a cíle, kam se má jednotka dostat. NavMesh agenti automaticky vyhledají nejkratší cestu k danému cíli.
4. NavMesh může být dynamicky aktualizován v průběhu hry, což umožňuje jednotkám reagovat na změny v prostředí a dynamicky plánovat svůj pohyb.



Obrázek 4.4: Vzhled připravené navigační sítě NavMesh. Modrá zóna je schůdná a NavMesh agenti se po ní mohou pohybovat, nezbarvená plocha je neprůchodná.

#### 4.3.4 Ekonomika hry

Hra bude obsahovat základní suroviny (dřevo, kámen), které budou hrát klíčovou roli v průběhu hry.

**Dřevo:** Dřevo lze získat těžbou stromů, které budou rozmístěné po herní mapě. Dřevo bude nezbytné pro výstavbu obranných struktur a budování celé osady. Jeho nedostatek může ztížit možnosti obrany a rozvoje osady.

**Kámen:** Kámen lze získat kopáním skal a hornin v okolí herního prostředí. Bude sloužit, jako stavební a vylepšující materiál pro obranné struktury a osadu. Nedostatek kamene může omezit hráčovy možnosti vylepšení obrany.

Všechny tyto suroviny budou hrát klíčovou roli v ekonomice a strategii hry. Hráči budou muset efektivně spravovat jejich zdroje a využívat je pro zajištění bezpečnosti a přežití své osady.

### Diagram tříd

Pro lepší představu modelu dat je zde uveden ER diagram, což je grafický nástroj používaný k vizualizaci a popisu struktury datového modelu.

[\[\[diagram dodělat\]\]](#)

## 4.4 Struktura projektu

Důležitou a často opomíjenou částí vývoje, je struktura samotného projektu, v této sekci je popsáno, jak byla vymyšlena organizace složek a souborů tak, aby byl jednoduše rozšířitelný a dobré se v něm orientovalo. Jednotlivé části projektu obsahují klíčové prvky této práce a jsou následovně rozděleny do jednotlivých složek. Tato sekce popisuje hlavní složky, do kterých je tento projekt rozdělen.

**[[přidat diagram struktury složek]]**

**Animations** Tato složka obsahuje soubory s animacemi postav a okolí ve hře. Animace jsou klíčovým prvkem pro pohyb a vizuální efekty herních postav a prostředí. V této složce se nachází více dalších podsložek, které dále rozdělují animace tak, aby usnadňovaly snadný přístup a správu animačních souborů.

**Graphics** Ve složce Graphics se nacházejí všechny grafické podklady použité v této hře, jako jsou textury, sprity, modely a další vizuální prvky. Grafické podklady jsou důležité pro vytváření vizuálního prostředí této hry a přispívají k celkovému vzhledu a atmosféře.

**Prefabs** Prefabs jsou předvytvořené herní objekty s přiřazenými skripty a dalšími vlastnostmi, které lze opakováně využívat ve hře. Tato složka obsahuje klíčové herní objekty, které jsou vytvořeny, konfigurovány předem a mohou být snadno použity v různých scénách hry. Příklad využití je například u stromů, které mají při inicializaci stejné vlastnosti, a tak mají vlastní prefab, který tyto informace společně se spritem obsahuje.

**Scenes** V této složce jsou definované herní scény, jako jsou menu, nebo samotná hlavní herní scéna hry. Každá scéna obsahuje specifické herní objekty, nastavení a logiku potřebnou pro danou část hry.

**ScriptableObjects** ScriptableObjects jsou objekty vytvořené pomocí skriptů, které dědí z třídy ScriptableObject vestavěné v Unity. Tyto objekty mohou uchovávat data, nastavení a další informace, které lze použít v různých částech hry. Hodí se například pro staviteľné objekty, které budou mít vždy stejné základní vlastnosti, jako je sprite, název, nebo kategorii, do které spadá, ke kterým si potom vývojář přidá potřebné skripty.

**Scripts** Ve složce scripts se nacházejí všechny skripty potřebné k běhu hry. Skripty obsahují herní logiku, ovladače, umělou inteligenci, UI interakce a další funkce potřebné k implementaci funkcí a chování vaší hry. Tato složka je dále rozdělená do podsložek, z důvodu ještě lepší organizace, přístupu a správy skriptů.

Při organizaci složek jsem zohlednil několik kritérií, která jsou důležitá pro efektivní správu a vývoj projektu. Při navrhování struktury složek jsem se zaměřil na funkční oddělení a jednoduchost formátu. Hlavní složky projektu jsou rozděleny podle typu obsahu a funkčnosti, co usnadňuje navigaci a správu projektu.

Dalším hlediskem, na které jsem se zaměřil, bylo oddělení logiky a dat, přičemž skripty obsahují herní logiku a funkce, zatímco ScriptableObjects slouží k uchování dat a nastavení nezávislých na konkrétních instancích herních objektů. Toto rozdělení pomáhá udržovat kód čistý a přehledný, což usnadňuje údržbu a správu projektu v průběhu vývoje.

# Kapitola 5

## Implementace

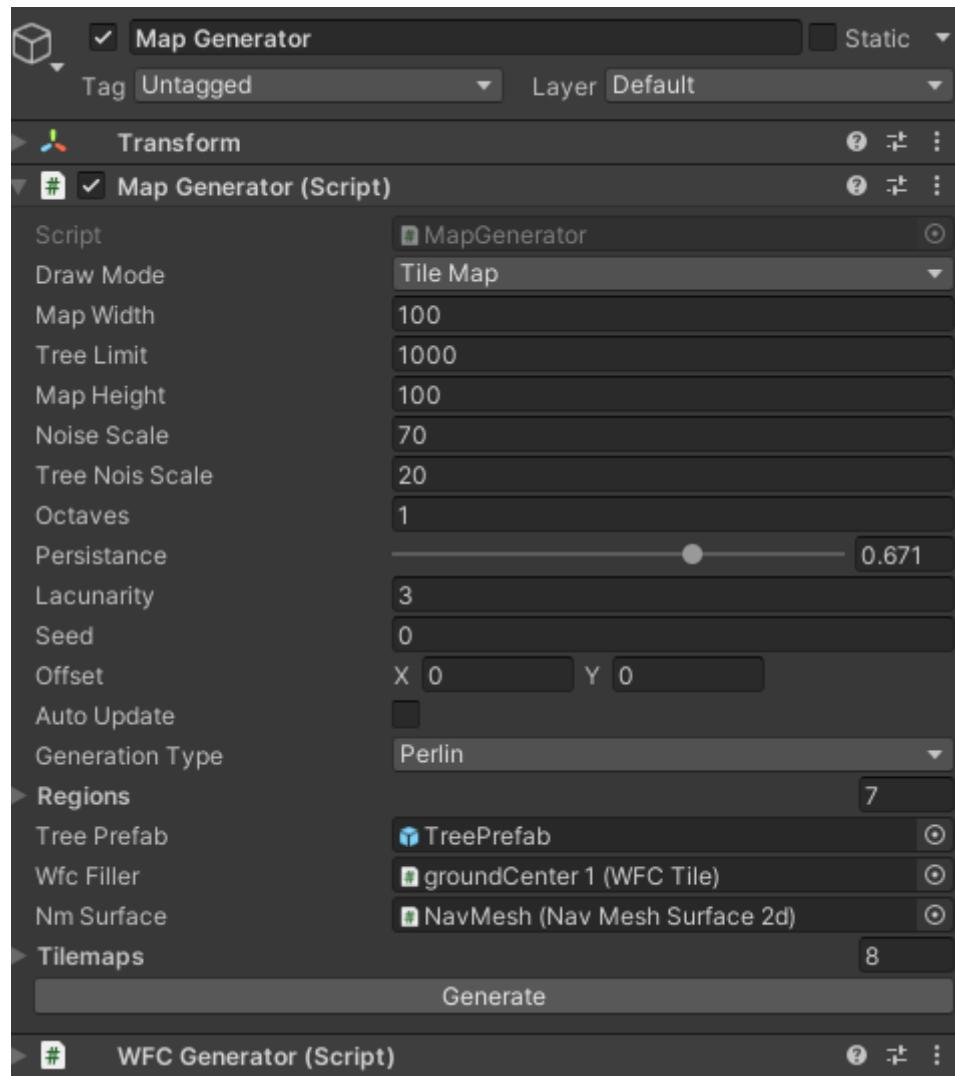
Tato kapitola se zaměřuje na detailní popis implementace skriptů a algoritmů používaných ve hře. V této části práce je popsána struktura a funkce jednotlivých skriptů a jejich využití v rámci celkového herního prostředí. Je zde vysvětlena, jak implementace základních prvků, jako jsou ovládací prvky a herní mechaniky, tak i složitější algoritmy pro procedurální generování obsahu, automatické vytváření herního menu a umělou inteligenci herních jednotek.

### 5.1 Implementace Perlinova šumu

Tato sekce se detailně zaměřuje na konkrétní implementaci Perlinova šumu v prostředí Unity pomocí skriptů v jazyce C#. Postup je vysvětlen krok za krokem a představuje klíčové prvky implementace, jako je využití Perlinova šumu, který je detailně popsán v sekci 3.3.2, a je implementován v knihovně *Mathf*. Skript *MapGenerator* nejprve vymaže původní hodnoty tilemap a následně se tilemapy zaplňují odpovídajícími tily.

Perlinův šum je generován pomocí knihovny *Mathf* v Unity. Implementace využívá metod *Mathf.PerlinNoise* k generování šumu s různými frekvencemi a amplitudami, čímž se vytváří plynulé, náhodné terény.

Ve skriptu *MapGenerator* je uživateli umožněno vybrat mezi generováním pomocí Perlinova šumu nebo algoritmu WFC. Tato volba závisí na potřebách a preferencích uživatele.



Obrázek 5.1: Herní objekt MapGenerator s přiřazenými skripty a nastavenými proměnnými.

### 5.1.1 Skript pro inicializaci a nastavení šumu

Skriptem, který má za úkol inicializovat generátor Perlinova šumu a umožňuje nastavení klíčových parametrů, jako jsou šířka a výška mapy, seed, přiblžení šumu, oktávy, apod. pro dosažení různorodých terénů, je pojmenován *MapGenerator*.

Jedná se o skript, který je přímo přiřazený hernímu objektu jménem *Map Generator*, ve kterém může vývojář pohodlně zadávat potřebné hodnoty pro generování šumu a samotného terénu. Tento skript byl implementován tak, aby i po jeho implementaci byl jednoduše upravitelný, proto lze přímo v editoru nastavovat hodnoty, jako je maximální počet stromů, šířka a výška mapy, zvolený algoritmus procedurálního generování, seed mapy a další.

Nejprve se volá externí metoda *GenerateNoiseMap()* z třídy *Noise* popsané níže 5.1.2, která generuje šumové dvoudimenzionální mapy pro terén, stromy a vegetaci. Tyto mapy slouží jako základní data pro následné umístění herních prvků na herní mapu. Po získání šumových map se vymaže obsah veškerých tilemap a tím se připraví na vkládání nového obsahu.

Pokud je zvolena metoda Wave Function Collapse (WFC), je vytvořen WFC generátor pomocí komponenty *WFFCGenerator*. Tento generátor je zodpovědný za vytváření mapy s ohledem na vzájemné interakce mezi buňkami a dalšími definovanými pravidly. Tento algoritmus a jeho implementace jsou podrobněji popsány v sekci 5.2.

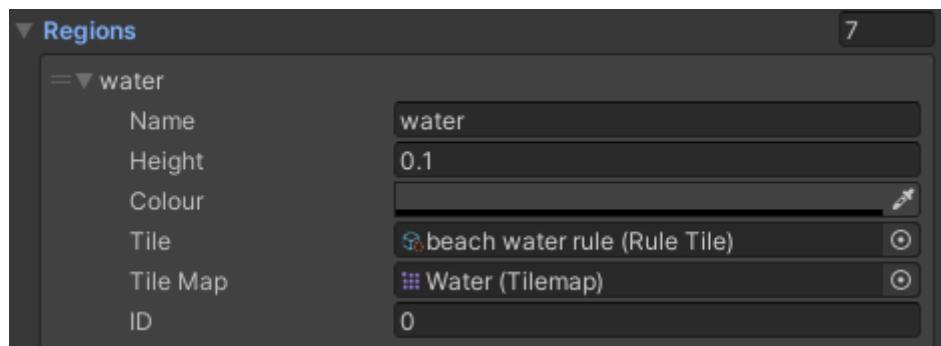
V opačném případě, tedy zvolení generování pomocí metody výškových map s použitím Perlinova šumu, se volá pomocná funkce *PlaceRegionTiles*, která pro každý bod mapy získá hodnotu šumu. Následuje porovnání podle definovaných regionů a umístění odpovídajícího tilu na příslušnou tilemapu

## Regiony

Regiony jsou tvořeny strukturou **TerrainType** definovanou následovně:

```
public struct TerrainType
{
    public string name;
    public float height;
    public Color colour;
    public TileBase tile;
    public Tilemap tileMap;
    public int ID;
}
```

Kde **name** je pojmenování daného regionu, **height** definuje horní limit pro šumovou hodnotu, to znamená že všechny hodnoty na mapě, které jsou nižší než tato hodnota a vyšší než hodnota předchozího regionu sem spadají. Proměnná **colour** je barva, která se používá pouze při nastavení generačního algoritmu a slouží pro testovací účely, jako vyobrazení obarvených bodů mapy. Hodnota **tile** je sprite, který se používá jako grafická podoba bodu mapy, může se jednat o sprite hory, vody, trávy a dalších. **tileMap** je tilemapa, na kterou se má nanášet **tile** a **ID** je identifikační číslo. Každý region má vlastní tilemapu, z důvodu následného generování NavMesh plochy více popsáné v sekci 4.3.3.



Obrázek 5.2: Ukázka jednoho z regionů s přiřazenými hodnotami.

### 5.1.2 Třída Noise

Třída Noise má na starosti vytváření samotného Perlinova šumu na základě zadaných vstupních hodnot. Tyto hodnoty následně upravuje v rámci potřebných mezí a volá funkci z knihovny *Mathf* jménem *PerlinNoise* pro každý bod z mapy. Tato hodnota je dále upravo-

vána amplitudou a dalšími hodnotami, po všech potřebných úpravách funkce vrací dvoudimenziorní pole hodnot pro každý bod mapy.

## 5.2 Wave function collapse

Další použitou generační metodou je algoritmus Wave Function Collapse (WFC). Tento algoritmus pracuje na ohodnocování buněk mapy a jejich následném *collapse*, což znamená vyhodnocení buňky a její nastavení na konkrétní hodnotu. Každá buňka má po inicializaci stejnou entropii, neboli počet možných tilů, kterých může nabývat. Samotný algoritmus funguje následovně:

1. Skript *MapGenerator* nejprve vytvoří podle zadané výšky a šířky mřížku buněk, které jsou typu *WFCCell* a mají v sobě uložený počet možností a entropii.
2. Po vytvoření všech buněk se každé buňce přiřazují její sousední buňky, z Von Neumannova pohledu, který je znázorněn na obrázku 3.1b.
3. V cyklu se hledá vždy buňka s nejnižší entropií, to znamená, že má nejméně možných tilů, kterých může nabývat, pokud existuje více s nejnižší entropií, vybere se náhodná z těchto buněk. Zároveň se při výběru vynechávají ty buňky, které mají nulovou entropii.
4. Pro tuto buňku se zavolá hlavní funkce skriptu *WFCGenerator* jménem *WaveFunction*.
5. Na buňku se zavolá *Collapse*, která vybere náhodný tile, tento výběr je vážený, aby některé tily měly větší šanci výběru.
6. Po nastavení nové hodnoty buňky je třeba zkонтrolovat okolní buňky a aktualizovat jejich možnosti a entropii. Pokud je nějaká buňka tímto způsobem aktualizována, tak je nutné zkонтrolovat i její sousední buňky.
7. tento proces se opakuje do té doby, dokud existuje alespoň jedna buňka, která ještě nebyla vyhodnocena.

Každá buňka *WFCCell* má na začátku nastavené všechny možné tily, kterých může nabývat. Tyto tily jsou typu *WFCTile* a obsahují informace jako jsou, váha, typ a povolené sousední tily, které se nastavují v editoru.

### 5.2.1 Skript WFCTile

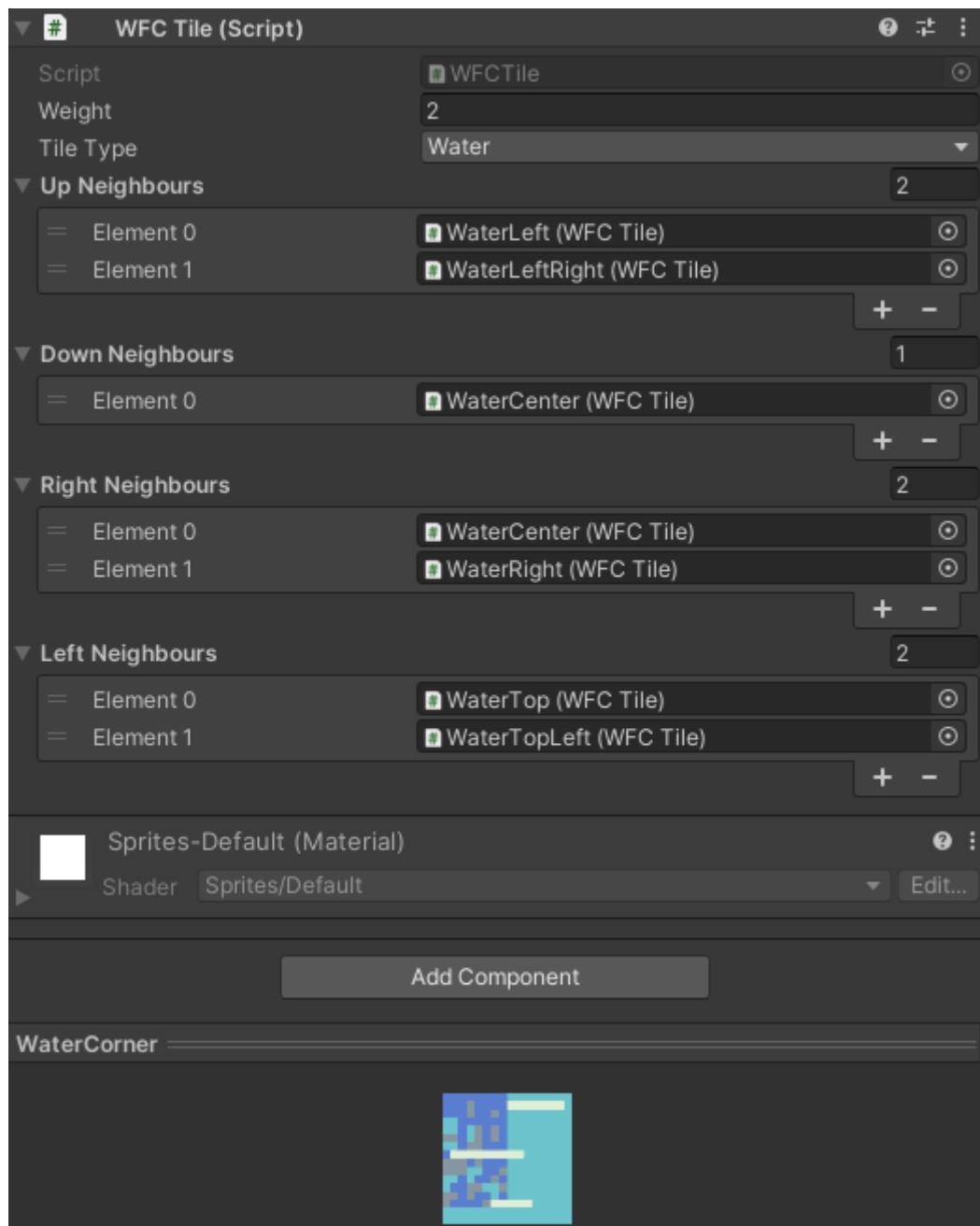
Slouží k definování chování a vlastností jednotlivých dlaždic (tile) v rámci algoritmu Wave Function Collapse (WFC). Jeho účel spočívá v udržování informací o sousedících dlaždicích a jejich vlastnostech, což je klíčové pro správné fungování algoritmu.

Zde je podrobnější popis jeho účelu:

**Reprezentace dlaždic:** Každá instance skriptu *WFCTile* reprezentuje jednu dlaždici v mapě, která může nabývat různých vlastností a typů v závislosti na kontextu generovaného prostředí.

**Definice sousedních dlaždic:** Skript uchovává informace o sousedních dlaždicích v různých směrech (nahoru, dolů, doleva, doprava). Tyto informace jsou zásadní pro správnou implementaci algoritmu WFC, protože určují možné hodnoty, které mohou sousední dlaždice nabývat.

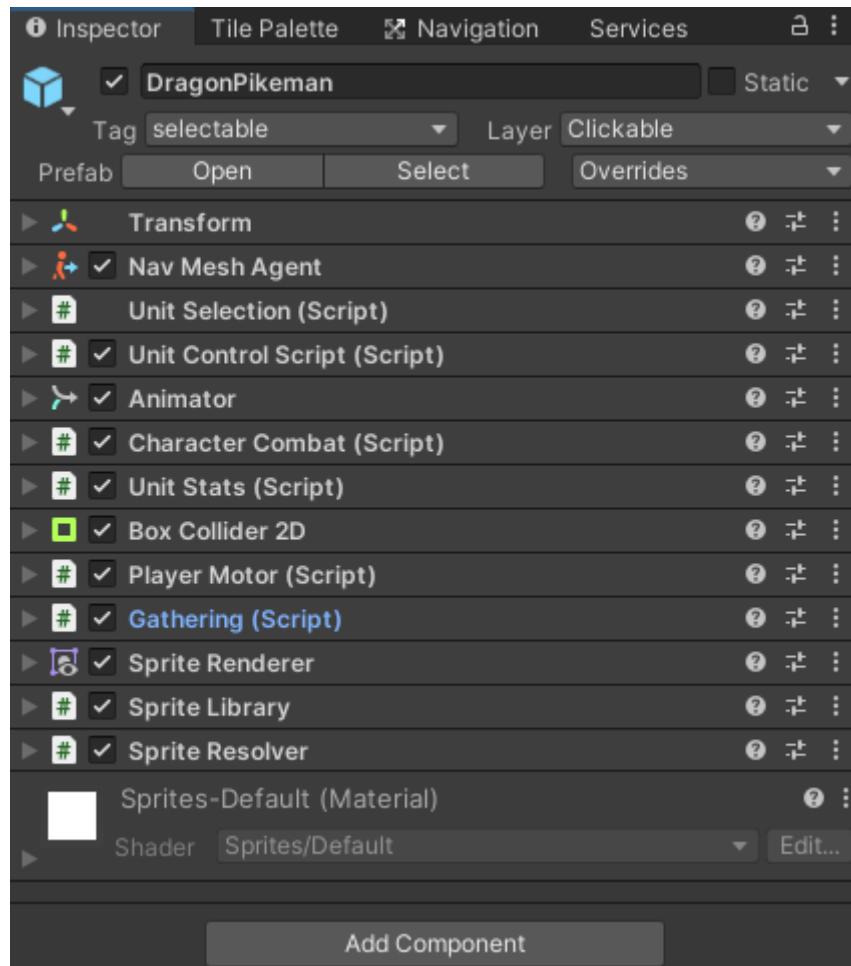
Celkově lze říci, že skript WFCTile má za úkol definovat jednotlivé dlaždice v rámci algoritmu WFC a zajišťuje správné fungování tohoto algoritmu pomocí udržování informací o vlastnostech dlaždic a jejich interakcích s okolními dlaždicemi.



Obrázek 5.3: Ukázka jednoho z mnoha tilů, každý tile má definované sousedy stejným způsobem.

## 5.3 Hráčovy jednotky

Tato kapitola se podrobněji zabývá implementací hráčových jednotek ve hře. Detailně se zde popisují jednotlivé skripty, které řídí chování jednotek, jejich interakci s okolím a výkon v boji.



Obrázek 5.4: Na obrázku jsou vidět komponenty hráčových jednotek.

### 5.3.1 Řízení a pohyb jednotek

Hlavním skriptem zajišťujícím řízení pohybu hráčových jednotek je **UnitControlScript**. Obsahuje metody pro interakci s uživatelským vstupem, pohyb k cílovým bodům, manipulaci s objekty a výběr cílů pro útok. Jeho hlavní funkce zahrnují:

**Zpracování uživatelského vstupu:** Skript sleduje uživatelský vstup a reaguje na něj, například detekuje kliknutí myší na herní plochu.

**Pohyb k bodům kliknutí** Po kliknutí myší na herní plochu skript řídí pohyb jednotky k danému cílovému bodu, či nepříteli.

**Interakce s objekty:** Jednotky mohou interagovat s různými objekty ve hře, například sbírat suroviny, nebo provádět útoky na nepřátelské jednotky.

Další důležitou součástí pohybu jednotky je skript **PlayerMotor**, jehož funkcionalita zahrnuje navigaci po herním světě. Pomocí komponenty **NavMeshAgent**, která je detailně popsána v sekci [4.3.3](#).

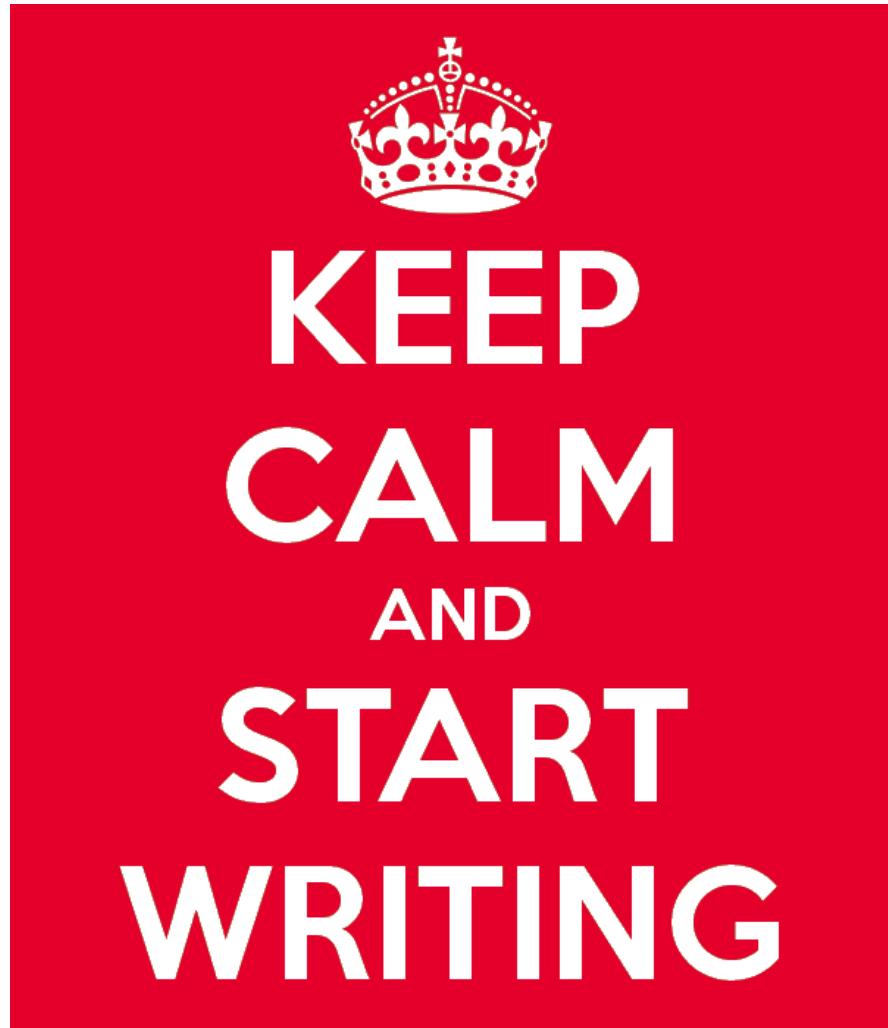
### 5.3.2 Ostatní třídy

Dalšími skripty, které jednotky využívají, jsou **UnitStats** a **Gathering**. Skript UnitStats je zodpovědný za správu statistik a vlastností hráčových jednotek v herním světě. Poskytuje klíčové informace o zdraví, útočné síle, rychlosti útoku, dosahu a dalších parametrech, které ovlivňují chování a výkon jednotek v boji.

Skript Gathering řídí funkce jednotky související se sběrem surovin v herním světě. Jednotka s tímto skriptem je schopna interagovat s herními objekty reprezentujícími suroviny a může je těžit. Tento skript úzce spolupracuje se skriptem **ResourceGatherScript**, který je více popsán v sekci [??](#).

### 5.3.3 Interakce a funkcionalita

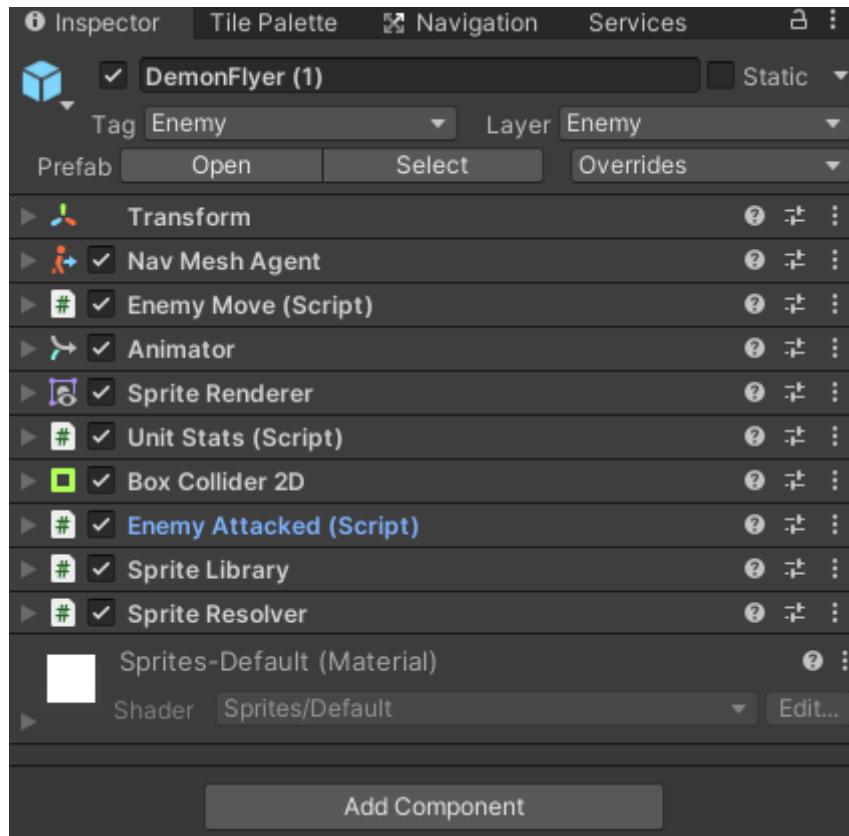
Interakce mezi jednotlivými skripty umožňuje jednotkám provádět různé akce v herním světě. Základní funkčnost je zaručena správnou provázaností a interakcí skriptů mezi sebou, ta je vyobrazena na diagramu [??](#)



Obrázek 5.5: Diagram zobrazující komunikaci skriptů UnitControlScript, Gathering, PlayerMotor, UnitStats a CharacterCombat.

## 5.4 Umělá inteligence nepřátel

Umělá inteligence (AI) nepřátel je jedním z nejdůležitějších prvků v této hře, přináší dynamiku a výzvy pro hráče. AI nepřátel je zodpovědná za jejich chování, rozhodování a reakce v průběhu hry.



Obrázek 5.6: Na obrázku jsou vidět komponenty nepřátelských jednotek.

Skript **EnemyMove** je jedním z kritických komponentů implementace tohoto chování. Tento skript řídí pohyb, detekci hráče a obstrukcí, pronásledování a útoky nepřátel. Díky tomuto skriptu jsou nepřátelé schopni inteligentně reagovat na hráčovy akce a poskytovat tak vyváženou a zábavnou herní zkušenosť. Skript také obsahuje řešení animací a rotací nepřítele směrem k hráči, což dodává hře vizuální efekty.

#### 5.4.1 Detekce hráče

Skript neustále skenuje okolí nepřítele a hledá hráče v dosahu. Provádí to pomocí metody **CheckForPlayer()**, která využívá vestavěnou funkci **Physics2D.OverlapCircleAll** k detekci všech hráčů v určené vzdálenosti nepřítele. Jakmile je v dosahu nalezen hráč, proměnná **chasedObject** je nastavena a nepřítel začne hráče pronásledovat.

#### 5.4.2 Pronásledování hráče

Pronásledování je prováděno metodou **HandleChase()**, která zjišťuje, zda je hráč v dosahu útoku, zda není překážka mezi nepřítelem a hráčem, a zda je hráč stále v dosahu sledování. Každá z těchto možností je dále popisována v následujících sekcích.

### Útok

Jakmile je hráč v dosahu útoku, nepřítel spustí útok pomocí metody **Attack()**. Při útoku nepřítel sleduje svou rychlosť útoku, aby neútočil příliš často, přičemž všechny statistiky

jednotky jsou uváděny ve skriptu **UnitStats.cs**. Pokud je hráč v dosahu útoku a je možné zaútočit, nepřítel způsobí škodu hráči, k tomu se používá funkce vestavěná v rozhraní **IAttackable** jménem **TakeDamage(int damage)**, kterou implementuje skript **UnitStats**. V případě, že je nepřítel střelec, používá útoky na dálku, tak vystřelí projektil směrem k hráči.

```
void Attack()
{
    agent.SetDestination(agent.transform.position);
    if (Time.time >= nextAttackEvent)
    {
        nextAttackEvent = Time.time + stats.attackSpeed;
        animator.SetTrigger("InRange");
        var target = chasedObject.GetComponent<IAttackable>();
        if (target != null)
        {
            if (stats.isRanged)
            {
                CreateBulletInDirection();
            }
            else
                target.TakeDamage(stats.attackDamage);
        }
    }
}
```

### Řešení překážek

Nepřítel se vyhýbá překážkám, pokud se nejedná o stavby hráče, v případě že se mezi hráčem a nepřítelem nachází zed, tak je novým cílem a nepřítel na ni útočí dokud ji nezničí.

### Rozhraní IAttackable

Toto rozhraní implementuje jak hráč, tak stavby. Jedná se o rozhraní, které obsahuje funkci **TakeDamage(int damage)**, Tímto způsobem je navržený proto, aby nepřítel nemusel zjišťovat a nahrávat pro hráče a zničitelnou stavbu jiné skripty. Díky IAttackable stačí pouze načíst tento interface a následně použít funkci TakeDamage(int damage).

## 5.5 Dynamický systém stavění

Stavební systém hry je založen na principu podobném jako vytváření herní mapy, hráči zde umisťují vybrané dlaždice na určená místa v herním světě na tilemapy. Tento systém umožňuje hráčům vytvářet a umisťovat stavby za běhu hry, přičemž každá stavba vyžaduje určité materiály a může být umístěna pouze na místa, kde ještě žádná jiná stavba nebyla postavena.

Stavební menu je dynamicky vytvářeno při spuštění hry, což umožňuje snadné rozšířování systému o nové kategorie a stavby v nich. Tento přístup byl zvolen pro zajištění škálovatelnosti a jednoduchosti přidávání nových prvků do hry.

Jednotlivé kategorie stavěných objektů jsou reprezentovány jako skriptovatelné objekty, tzv. *scriptable objects*. Tyto kategorie jsou viditelné ve hře v dolní části obrazovky a jsou načítány dynamicky. Každá kategorie uchovává tilemapu, na kterou se budou stavby pokládat, a také *PlaceType*, což je enum, který určuje způsob, jakým lze danou stavbu umístit (např. jednotlivě, v řadě, do obdélníku atd.).

Skriptovatelný objekt **BuildableObjectBase** je abstraktní třída, která reprezentuje jednotlivé stavitelné objekty ve hře. Každá instance této třídy obsahuje informace a vlastnosti specifické pro daný typ budovy, jako je kategorie, typ umístění, prefabrikát budovy a požadované materiály pro stavbu.

### 5.5.1 Vytváření stavebního menu

**BuildingHUD** je třída, která řídí vytváření uživatelského rozhraní (UI) pro stavitelské menu ve hře. Tento skript je zodpovědný za vytváření kategorií budov a položek v těchto kategoriích, aby hráč mohl jednoduše vybírat, které budovy chce postavit. Tento skript umožňuje:

**Dynamické vytváření UI** Skript automaticky vytváří tlačítka pro různé kategorie budov a zobrazuje je ve stavitelském menu. To umožňuje hráčům snadno procházet různé typy budov, které mohou postavit.

**Dynamické načítání dat** Data o stavitelných budovách jsou načítána ze speciálních souborů ve složce "Resources/ScriptableObjects/Buildables". To umožňuje jednoduché přidávání nových budov do hry bez nutnosti úpravy kódu.

Tento skript je navržený tak, aby bylo možné dynamicky přidávat nové kategorie a položky, a to bez nutnosti jakékoli úpravy kódu. Na obrázku 4.3 je v levém dolním rohu vidět stavební menu.

### 5.5.2 Používání stavebního menu

Klíčovou třídou odpovědnou za tvorbu a umisťování stavebních objektů ve hře je **BuildingCreator**. Tato třída je navržena tak, aby byla flexibilní a snadno rozšiřitelná pro budoucí vývoj.

Jedním z klíčových prvků BuildingCreator je jeho schopnost dynamicky vytvářet a umisťovat stavební objekty na herní mapu. To umožňuje hráčům vybírat z různých typů budov a umisťovat je na vybraná místa pomocí myši. Třída také poskytuje náhledovou mapu, která hráčům ukazuje, kam bude budova umístěna, a zda je to možné z hlediska terénu, či surovin.

Další klíčovou funkcí je správa kategorií stavebních objektů. Objekty jsou rozděleny do kategorií, po rozkliknutí se zobrazí podmenu, ze kterého lze vybrat konkrétní stavbu. Kategorie jsou definovány pomocí skriptovatelných objektů popsaných v sekci 5.5. BuildingCreator také zajišťuje, aby hráči nemohli umístit stavební objekty na zakázaná místa nebo na místa, kde již existuje jiná stavba.

Každá budova je reprezentována prefabrikátem, který obsahuje potřebné komponenty jako *collider2D*, komponentu na detekci kolizí, skript odpovídající dané stavbě a popřípadě *NavMeshObstacle*, což je komponenta, která v NavMesh povrchu vytváří neprůchodnou oblast.

## Metoda DrawBounds

Metoda **DrawBounds** se stará o vykreslení obdélníkových a lineárních struktur na mapě a zároveň provádí kontrolu dostupnosti zdrojů a umístění na mapě. Metoda kombinuje výpočet hranic oblasti, kontrolu a aktualizaci stavu surovin a umisťování prefabrikátů na mapu. Tato metoda je klíčovou součástí procesu tvorby a umisťování stavebních objektů a ukazuje, jakým způsobem se zde řeší komplexní problémy spojené s tímto procesem. Zde je ukázka kódu:

```
if (previewMode)
{
    map.SetTile(new Vector3Int(posX, posY, 0), tileBase);

    if (IsForbidden(new Vector3Int(posX, posY, 0)) || NotEnoughResources(requiredRes
    {
        if (!IsSameTilemap(new Vector3Int(posX, posY, 0), selectedObject.Category
            map.SetColor(new Vector3Int(posX, posY, 0), Color.red);
    }
}
else
{
    Vector3Int targetPosition = new Vector3Int(posX, posY, 0);
    previewMap.SetTile(targetPosition, null);
    if (IsBuildablePosition)
    {
        resourceMenu.UpdateAmmount(-requiredResources.ammount, requiredResources
        if (requiredResourcesPreview.ContainsKey(requiredResources.itemType))
            requiredResourcesPreview[requiredResources.itemType] -= requiredR

        map.SetTile(targetPosition, tileBase);
        GameObject item = Instantiate(selectedObject.Prefab, targetPosition, Quat
        item.transform.parent = categoryParent.transform;
    }
}
}
```

Tento úryvek ukazuje, jak se dynamicky pracuje s rozložením zdrojů a umístěním stavby. Prochází se celý obdélníkový rozměr stavby a provádí se analýza dostupných zdrojů pro stavbu. Pokud jsou zdroje dostupné a na cestě není překážka, stavba se umístí a zdroje jsou odebrány. V opačném případě se dokončí pouze ty části, kde to je možné a v náhledu se ty části které nebudou postaveny zabarví červeně.

## 5.6 Náhodné události

Náhodné události představují důležitý prvek v herním designu, který přináší do hry dynamiku, variabilitu a nepředvídatelnost. Jejich implementace je klíčová pro zajímavý a zábavný herní zážitek. Na obrázku 5.7 je vyobrazené informativní okno při spuštění náhodné události Invasion.



Obrázek 5.7: Snímek obrazovky v moment kdy nastala náhodná událost, v tomto případě konkrétně Invasion, která vytvoří nepřátele útočící na hráče.

### 5.6.1 Třída EventManager

Slouží jako centrální prvek pro správu náhodných událostí v herním prostředí. Její hlavní funkcí je spravovat seznam dostupných událostí a umožňovat jejich vyvolání v reakci na určité podmínky nebo události ve hře. Implementace třídy EventManager zahrnuje následující klíčové prvky:

**Správa seznamu událostí:** Třída EventManager udržuje seznam všech dostupných událostí, které mohou být vyvolány v průběhu hry. Tyto události mohou zahrnovat různé situace, jako jsou útoky nepřátel, získání nových jednotek atd.

**Generování náhodných událostí:** EventManager obsahuje metody pro generování náhodných událostí z dostupného seznamu událostí. Tato funkcionality umožňuje vytvoření dynamického a nepředvídatelného herního prostředí, které udržuje hráče zapojené a zaujaté.

Každý GameEvent má atribut GameEventAttribute, díky kterému lze dynamicky a automaticky pomocí reflexe načítat a inicializovat všechny možné události ze skriptu EventManager, pro což byla vytvořena následující funkce:

```
void LoadEvents()
{
    var eventTypes = Assembly.GetExecutingAssembly().GetTypes()
        .Where(type => type.GetCustomAttributes(typeof(GameEventAttribute), true)

    foreach (var eventType in eventTypes)
    {
```

```

        var eventInstance = Activator.CreateInstance(eventType) as IGameEvent;
        if (eventInstance != null)
        {
            events.Add(eventInstance);
        }
    }
}

```

Tato funkce slouží k načtení dostupných událostí do paměti z aktuálně běžícího assembly (souboru s kódem aplikace) a jejich uložení do kolekce událostí. První řádek **Assembly.GetExecutingAssembly().GetTypes()** vrací aktuálně běžící assembly, ve kterém je tato funkce vykonávána a všechny typy v jeho rámci. Následně se pomocí Linq filtrují všechny typy událostí, na základě toho, zda mají atribut **GameEventAttribute**. Následuje iterace přes všechny nalezené typy událostí a instanciaci daného typu události za pomoci reflexe. Reflexe umožňuje manipulaci s typy a objekty za běhu programu. Zde je použita k vytvoření instance třídy a přetytování na **IGameEvent**. Nakonec se každá událost přidává do seznamu, ze kterého se následovně náhodně vybírají.

Tato třída navíc obsahuje seznam prefabů nepřátelských jednotek, zařizuje manipulaci s informativním oknem a má pomocné funkce díky kterým mohou GameEventy načítat potřebné hodnoty.

### 5.6.2 Rozhraní IGameEvent

Definuje základní strukturu a metody, které musí být implementovány v jednotlivých náhodných událostech. Toto rozhraní poskytuje framework pro vytváření a správu konkrétních událostí v rámci herního prostředí. Klíčové prvky implementace rozhraní IGameEvent zahrnují:

**Metody pro inicializaci a spuštění události:** Rozhraní obsahuje metody *InitEvent()* a *StartEvent()*, které slouží k inicializaci a spuštění konkrétní události v rámci herního světa. Tyto metody umožňují přípravu a spuštění události v reakci na určité podmínky ve hře.

**Metoda pro aktualizaci události:** Rozhraní definuje metodu *UpdateEvent()*, která slouží k aktualizaci stavu události v průběhu hry. Tato metoda umožňuje monitorování a řízení průběhu události a provádění potřebných akcí v závislosti na aktuálním stavu hry.

**Metoda pro ukončení události:** Rozhraní obsahuje metodu *EndEvent()*, která slouží k ukončení dané události po jejím dokončení, nebo po dosažení určitých podmínek. Tato metoda umožňuje vyčištění prostředků a provedení konečných akcí spojených s danou událostí. Metoda *EndEvent()* navíc vrací hodnotu typu *WindowInfo*, což jsou hodnoty, které jsou potřebné pro zobrazení informativního okna hráči po dokončení náhodné události.

# Literatura

- [1] ANDRADE, A. Game engines: a survey. *EAI Endorsed Transactions on Serious Games*. EAI. Listopad 2015, sv. 2, č. 6. DOI: 10.4108/eai.5-11-2015.150615.
- [2] ANJYO, K.-i. A Simple Spectral Approach to Stochastic Modelling for Natural Objects. In: *EG 1988-Technical Papers*. Eurographics Association, 1988. DOI: 10.2312/egtp.19881023. ISSN 1017-4656.
- [3] BARNSLEY, M. F., DEVANEY, R. L., MANDELBROT, B. B., PEITGEN, H.-O., SAUPE, D. et al. Algorithms for random fractals. *The science of fractal images*. Springer. 1988, s. 71–136.
- [4] BENEŠ, B. A stable modeling of large plant ecosystems. In: *Proceedings of the International Conference on Computer Vision and Graphics*. 2002, s. 94–101.
- [5] COOK, R. L. a DEROSE, T. Wavelet noise. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. jul 2005, sv. 24, č. 3, s. 803–811. DOI: 10.1145/1073204.1073264. ISSN 0730-0301. Dostupné z: <https://doi.org/10.1145/1073204.1073264>.
- [6] DHARIWAL, P., JUN, H., PAYNE, C., KIM, J. W., RADFORD, A. et al. Jukebox: A Generative Model for Music. *ArXiv*. 2020, abs/2005.00341. Dostupné z: <https://api.semanticscholar.org/CorpusID:218470180>.
- [7] FOURNIER, A., FUSSELL, D. a CARPENTER, L. Computer rendering of stochastic models. In: *Seminal Graphics: Pioneering Efforts That Shaped the Field, Volume 1*. New York, NY, USA: Association for Computing Machinery, 1998, s. 189–202. ISBN 158113052X. Dostupné z: <https://doi.org/10.1145/280811.280993>.
- [8] FREIKNECHT, J. *Procedural content generation for games*. Mannheim, 2021. Disertační práce. Dostupné z: <https://madoc.bib.uni-mannheim.de/59000/>.
- [9] GAMITO, M. N. Procedural Landscapes with Overhangs. In:. 2001. Dostupné z: <https://api.semanticscholar.org/CorpusID:18543187>.
- [10] GOLDBERG, A., ZWICKER, M. a DURAND, F. Anisotropic noise. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. aug 2008, sv. 27, č. 3, s. 1–8. DOI: 10.1145/1360612.1360653. ISSN 0730-0301. Dostupné z: <https://doi.org/10.1145/1360612.1360653>.
- [11] GONG, Y. A survey on the modeling and applications of cellular automata theory. *IOP Conference Series: Materials Science and Engineering*. IOP Publishing. sep 2017, sv. 242, č. 1, s. 012106. DOI: 10.1088/1757-899X/242/1/012106. Dostupné z: <https://dx.doi.org/10.1088/1757-899X/242/1/012106>.

- [12] GUO, J., JIANG, H., BENES, B., DEUSSEN, O., ZHANG, X. et al. Inverse Procedural Modeling of Branching Structures by Inferring L-Systems. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. jun 2020, sv. 39, č. 5. DOI: 10.1145/3394105. ISSN 0730-0301. Dostupné z: <https://doi.org/10.1145/3394105>.
- [13] HART, J. C. Perlin noise pixel shaders. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. New York, NY, USA: Association for Computing Machinery, 2001, s. 87–94. HWWS ’01. DOI: 10.1145/383507.383531. ISBN 158113407X. Dostupné z: <https://doi.org/10.1145/383507.383531>.
- [14] HART, J. C., CARR, N., KAMEYA, M., TIBBITTS, S. A. a COLEMAN, T. J. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. 1999, s. 45–53.
- [15] HENDRIKX, M., MEIJER, S., VAN DER VELDEN, J. a IOSUP, A. Procedural Content Generation for Games: A Survey. New York, NY, USA: Association for Computing Machinery. feb 2013, sv. 9, č. 1. DOI: 10.1145/2422956.2422957. ISSN 1551-6857. Dostupné z: <https://doi.org/10.1145/2422956.2422957>.
- [16] IVSON, P., TOLEDO, R. a GATTASS, M. Solid height-map sets: modeling and visualization. In: červen 2008, s. 359–365. DOI: 10.1145/1364901.1364953.
- [17] JOHNSON, L., YANNAKAKIS, G. a TOGELIUS, J. Cellular automata for real-time generation of. Září 2010. DOI: 10.1145/1814256.1814266.
- [18] KENSLER, A. E. 1 Better Gradient Noise. In: 2008. Dostupné z: <https://api.semanticscholar.org/CorpusID:15788995>.
- [19] LAGAE, A., LEFEBVRE, S., COOK, R., DEROSSE, T., DRETTAKIS, G. et al. A Survey of Procedural Noise Functions. *Computer Graphics Forum*. 2010, sv. 29, č. 8, s. 2579–2600. DOI: <https://doi.org/10.1111/j.1467-8659.2010.01827.x>. Dostupné z: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2010.01827.x>.
- [20] LAGAE, A., LEFEBVRE, S., DRETTAKIS, G. a DUTRÉ, P. Procedural noise using sparse Gabor convolution. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. jul 2009, sv. 28, č. 3. DOI: 10.1145/1531326.1531360. ISSN 0730-0301. Dostupné z: <https://doi.org/10.1145/1531326.1531360>.
- [21] LEWIS, J. P. Methods for stochastic spectral synthesis. In: *Proceedings on Graphics Interface ’86/Vision Interface ’86*. CAN: Canadian Information Processing Society, 1986, s. 173–179.
- [22] LEWIS, J. P. Generalized stochastic subdivision. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery. jul 1987, sv. 6, č. 3, s. 167–190. DOI: 10.1145/35068.35069. ISSN 0730-0301. Dostupné z: <https://doi.org/10.1145/35068.35069>.
- [23] LEWIS, J. P. Algorithms for solid noise synthesis. In: *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1989, s. 263–270. SIGGRAPH ’89. DOI:

10.1145/74333.74360. ISBN 0897913124. Dostupné z:  
<https://doi.org/10.1145/74333.74360>.

- [24] LEWIS, J.-P. Texture synthesis for digital painting. *SIGGRAPH Comput. Graph.* New York, NY, USA: Association for Computing Machinery. jan 1984, sv. 18, č. 3, s. 245–252. DOI: 10.1145/964965.808605. ISSN 0097-8930. Dostupné z: <https://doi.org/10.1145/964965.808605>.
- [25] LINDENMAYER, A. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of theoretical biology*. Elsevier. 1968, sv. 18, č. 3, s. 280–299.
- [26] LINDENMAYER, A. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*. 1968, sv. 18, č. 3, s. 280–299. DOI: [https://doi.org/10.1016/0022-5193\(68\)90079-9](https://doi.org/10.1016/0022-5193(68)90079-9). ISSN 0022-5193. Dostupné z: <https://www.sciencedirect.com/science/article/pii/0022519368900799>.
- [27] LIU, J., SNODGRASS, S., KHALIFA, A., RISI, S., YANNAKAKIS, G. N. et al. Deep learning for procedural content generation. *Neural Computing and Applications*. jan 2021, sv. 33, č. 1, s. 19–37. DOI: 10.1007/s00521-020-05383-8. ISSN 1433-3058. Dostupné z: <https://doi.org/10.1007/s00521-020-05383-8>.
- [28] MILLER, G. S. P. The Definition and Rendering of Terrain Maps. New York, NY, USA: Association for Computing Machinery. aug 1986, sv. 20, č. 4, s. 39–48. DOI: 10.1145/15886.15890. ISSN 0097-8930. Dostupné z: <https://doi.org/10.1145/15886.15890>.
- [29] MĚCH, R. a PRUSINKIEWICZ, P. Visual models of plants interacting with their environment. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1996, s. 397–410. SIGGRAPH '96. DOI: 10.1145/237170.237279. ISBN 0897917464. Dostupné z: <https://doi.org/10.1145/237170.237279>.
- [30] NEWLANDS, C. a ZAUNER, K.-P. *Procedural Generation and Rendering of Realistic, Navigable Forest Environments: An Open-Source Tool*. 2022.
- [31] NILSON, B. a SÖDERBERG, M. Game Engine Architecture. *Mälardalen University*. 2007.
- [32] OLANO, M. Modified noise for evaluation on graphics hardware. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. New York, NY, USA: Association for Computing Machinery, 2005, s. 105–110. HWWS '05. DOI: 10.1145/1071866.1071883. ISBN 1595930868. Dostupné z: <https://doi.org/10.1145/1071866.1071883>.
- [33] PERLIN, K. An image synthesizer. *SIGGRAPH Comput. Graph.* New York, NY, USA: Association for Computing Machinery. jul 1985, sv. 19, č. 3, s. 287–296. DOI: 10.1145/325165.325247. ISSN 0097-8930. Dostupné z: <https://doi.org/10.1145/325165.325247>.
- [34] PERLIN, K. Improving noise. *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 2002. Dostupné z: <https://api.semanticscholar.org/CorpusID:207550606>.

- [35] PEYTAVIE, A., GALIN, E., GROSJEAN, J. a MÉRILLOU, S. Arches: a Framework for Modeling Complex Terrains. *Computer Graphics Forum*. Duben 2009, sv. 28, s. 457 – 467. DOI: 10.1111/j.1467-8659.2009.01385.x.
- [36] PRUSINKIEWICZ, P. Graphical applications of L-systems. In: *Proceedings of graphics interface*. 1986, sv. 86, č. 86, s. 247–253.
- [37] PRUSINKIEWICZ, P., HAMMEL, M. S. a MJOLSNESS, E. Animation of plant development. In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1993, s. 351–360. SIGGRAPH '93. DOI: 10.1145/166117.166161. ISBN 0897916018. Dostupné z: <https://doi.org/10.1145/166117.166161>.
- [38] PRUSINKIEWICZ, P., JAMES, M. a MĚCH, R. Synthetic topiary. In: *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1994, s. 351–358. SIGGRAPH '94. DOI: 10.1145/192161.192254. ISBN 0897916670. Dostupné z: <https://doi.org/10.1145/192161.192254>.
- [39] PRUSINKIEWICZ, P. a LINDENMAYER, A. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.
- [40] RODEN, T. a PARBERRY, I. *From Artistry to Automation: A Structured Methodology for Procedural Content Creation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. 151–156 s. ISBN 978-3-540-28643-1.
- [41] SMELIK, R., KRAKER, K. J. de, GROENEWEGEN, S., TUTENEL, T. a BIDARRA, R. A Survey of Procedural Methods for Terrain Modelling. In:. červen 2009.
- [42] TECHNOLOGIES, U. *Compare unity plans: Personal, pro, enterprise, industry*. 2024.
- [43] VOHERA, C., CHHEDA, H., CHOUHAN, D., DESAI, A. a JAIN, V. Game engine architecture and comparative study of different game engines. In: IEEE. *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. 2021, s. 1–6.
- [44] WIJK, J. J. van. Spot noise texture synthesis for data visualization. *SIGGRAPH Comput. Graph.* New York, NY, USA: Association for Computing Machinery. jul 1991, sv. 25, č. 4, s. 309–318. DOI: 10.1145/127719.122751. ISSN 0097-8930. Dostupné z: <https://doi.org/10.1145/127719.122751>.
- [45] WORLEY, S. A cellular texture basis function. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, s. 291–294.