VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

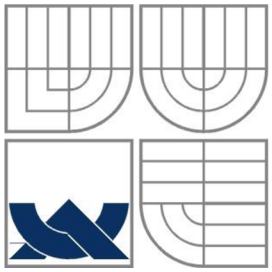
GENEROVÁNÍ MAPY A HLEDÁNÍ CESTY V MAPĚ PRO STRATEGICKÉ HRY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

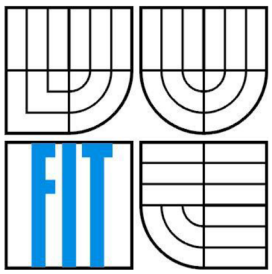
AUTOR PRÁCE
AUTHOR

ŠTĚPÁN KARÁSEK

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ MAPY A HLEDÁNÍ CESTY V MAPĚ PRO STRATEGICKÉ HRY

MAP GENERATOR AND PATH SEARCH IN THE MAP FOR THE STRATEGY GAMES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ŠTĚPÁN KARÁSEK

VEDOUCÍ PRÁCE
SUPERVISOR

ING. FILIP ORSÁG, Ph.D.

BRNO 2014

Abstrakt

Tato práce se skládá ze dvou částí. V první části se zabývá procedurálním generováním mapy se zaměřením na tvorbu terénu. Nejdříve jsou rozebrány principy generování terénu, použitelné algoritmy a způsoby generování dalších krajinných prvků mapy. Generování terénu za použití algoritmu diamond-square je následně implementováno a začleněno do existujícího herního enginu.

Druhá část práce pojednává o problému hledání cesty v mapě, jsou zde popsány použitelné algoritmy a jeden z nich podrobněji vysvětlen. Dále se v práci řeší úprava nalezené cesty do přirozenějších tvarů a nastiňují se možnosti hierarchických algoritmů pro hledání cesty. Algoritmus A* pro hledání cesty je rovněž implementován a začleněn do existujícího herního enginu. Funkčnost obou algoritmů je prezentována v interaktivní aplikaci s grafickým rozhraním.

Abstract

This thesis consists of two parts. The first part deals with procedural map generation aiming at terrain generation. For the most part, the bases of terrain generation, usable algorithms and the ways of generating other landscape elements are followed up. A terrain generator using diamond-square algorithm is then implemented and integrated into the existing game engine.

The second part of the thesis analyses the pathfinding problem, describes usable algorithms and one of them in detail. Furthemore, the post-processing of the path is being solved and possibilities of hierarchical pathfinding algorithms are presented. Pathfinding using A* algorithm is also implemented and integrated into existing game engine. Finally, functioanality of both the algorithms is demonstrated in an interactive application.

Klíčová slova

Generování mapy, fraktály, diamond-square algoritmus, regulace generovaného terénu, hledání cesty v mapě, algoritmus A*, vyhlazování cesty, Java

Keywords

Map generator, fractals, diamond-square algorithm, controled terrain generation, pathfinding, A* algorithm, path smoothing, Java

Citace

Karásek Štěpán: Generování mapy a hledání cesty v mapě pro strategické hry, bakalářská práce, Brno, FIT VUT v Brně, 2014

Generování mapy a hledání cesty v mapě pro strategické hry

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Filipa Orsága, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Štěpán Karásek
19. května 2014

Poděkování

Rád bych poděkoval svému vedoucímu Ing. Filipu Orságovi, Ph.D. za vedení a užitečné rady poskytnuté při řešení této práce.

Obsah

1 Úvod 2
2 Procedurální generování mapy 3
2.1 Fraktálová geometrie 3
2.2 Přístup ke generování terénu 4
2.2.1 Perlin noise 5
2.2.2 Simplex noise 5
2.2.3 Algoritmus Midpoint-Displacement 5
2.2.4 Algoritmus Diamond-Square 6
2.2.5 Genetické algoritmy 6
2.2.6 Worley noise 7
2.2.7 Hill algorithm 7
2.3 Diamond-square algoritmus 7
2.4 Regulace generovaného terénu 9
2.5 Dotváření povrchu terénu 9
2.6 Generování dalších úrovní mapy 10
3 Hledání cesty v mapě 12
3.1 Reprezentace problému 12
3.2 Metody prohledávání grafů 12
3.2.1 Primitivní metody 13
3.2.2 Breadth First Search 13
3.2.3 Dijkstrův algoritmus 13
3.2.4 Best First Search algoritmy 13
3.3 Algoritmus A* 14
3.3.1 Heuristické funkce 16
3.3.2 Optimálnost algoritmu A* 16
3.4 Víceúrovňové prohledávání prostoru a post-processing nalezené cesty 17
3.5 Hierarchické algoritmy pro hledání cesty 19
4 Implementace vybraných algoritmů 20
4.1 Generování terénu 20
4.2 Hledání cesty přes mapu 22
4.2.1 Post-processing nalezené cesty 25
5 Shrnutí výsledků 27
5.1 Generování terénu 27
5.2 Hledání cesty přes vygenerovanou mapu 29
6 Závěr 31

1 Úvod

V dnešní době neustále roste výkonost počítačových sestav. Počítače jsou schopné ukládat čím dál tím více dat a více jich i zprácovávat. Zároveň stoupá i poptávka uživatelů po něčem novém a lepším. Ruku v ruce s tím se předhání také vývojáři softwaru snažící se poskytnout lidem co nejkomplexnější a mnohdy i co nejvěrnější virtuální světy. Ty se mohou velice lišit podle účelu, ke kterému slouží. V jedné věci se však shodují. Uživatelům se líbí volnost a jelikož je člověk tvor zvídavý, tak i možnost objevovat něco nového. Proto se vývojáři snaží virtuální světy rozšířit, aby poskytovaly co největší možnosti. Větší a komplexnější virtuání světy však potřebují nejen značné množství paměti, na které je potřeba uložit všechna data, ale i spoustu času vývojářů, kteří musí takový svět vytvořit.

Nevýhodou je také neměnnost ručně vytvořeného světa. Ten je jednou vytvořen a zůstává pokaždé stejný, má přesně dané hranice a uživatel po jisté době může dojít k pocitu, že zde není nic zajímavého a přestane používat daný software. Navíc, v dnešní době je možné propojení uživatelů přes internet a prozkoumávání jednoho virtuálního světa dohromady. S tím nadále roste potřeba vytvořit virtuální svět ještě větší, aby byl schopný uspokojit potřeby velkého množství uživatelů. Vytvoření takového světa, stejně tak jako vytváření velkého množství různých map za účelem větší variability, je velice pracné a tím i nákladné, čímž vzniká potřeba automatizovat tento proces.

Prvním cílem této práce bude procedurální generování terénu zaměřené na strategické hry. Z toho důvodu nebude brán takový zřetel na vizuální realističnost, jako spíše na smysluplnost vytvořené mapy, její použitelnosti z hlediska využití strategických her a celkovou konzistenci vytvořeného terénu.

Další část této práce vychází částečně z té předchozí. Bude zaměřená na vyhledávání cesty přes takto vytvořenou mapu. Hledání cesty přes mapu se může zdát na první pohled relativně triviální problém. To však platí pouze do doby, kdy je velikost a členitost mapy natolik nízká, že je lidský mozek schopný tyto informace zpracovat. K tomuto účelu navíc využívá vizuální informace a svojí výbornou schopnost je správně pochopit a použít.

Algoritmické hledání cesty v mapě je více podobné procházení bludiště, kdy nevíme, jestli jdeme správnou cestou a jestli se zrovna nevydáme slepou uličkou. Jedinou možností, jak to zjistit, je vyzkoušet všechny možnosti. Ve větším měřítku se proto může jednat o velmi výpočetně náročný proces. Zvláště, jedná-li se o strategické hry, ve kterých se většinou pohybuje větší množství objektů. V tomto směru je potřeba se zaměřit nejen na kvalitu výsledků algoritmu, co se týče nalezení cesty, ale i na rychlost daného řešení.

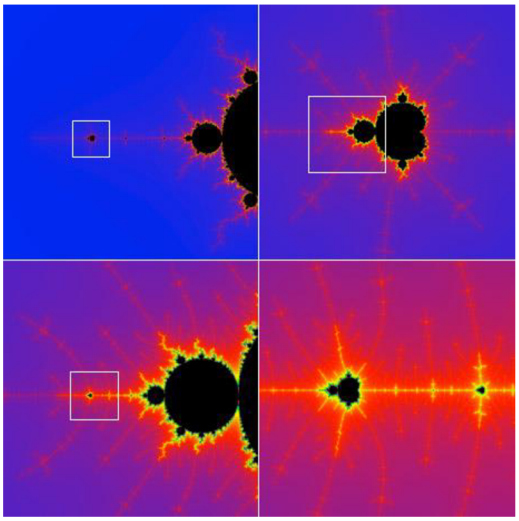
Cílem této práce je, kromě seznámení se dostupnými metodami pro procedurální generování terénu a hledáním cesty tímto vygenerovaným terénem, i implementace vybraných metod a jejich začlenění do existujícího herního enginu.

2 Procedurální generování mapy

2.1 Fraktálová geometrie

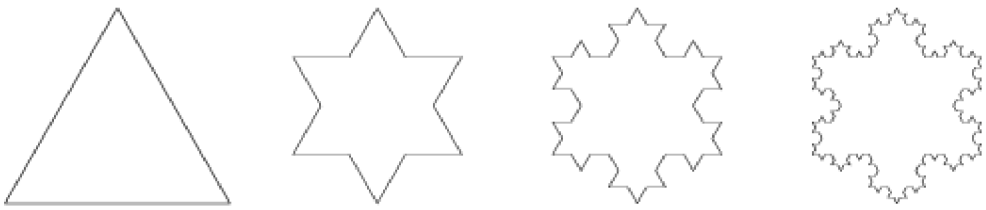
Jednou z prvních věcí, která člověka napadne, když pozoruje přírodu, je její ohromná komplexnost a rozmanitost. Přesto, že se může zdát, že lidské výtvory jsou velmi složité a na vysoké úrovni, v porovnání s produkty přírody stále zaostávají. Po dlouhou dobu si lidé nedokázali poradit s tím, odkud se tato komplexnost bere a jakým způsobem ji matematicky popsat, jestli je to vůbec možné.

Do tohoto problému se vložil polský matematik Benoit Mandelbrot (1924-2010), který pracoval na množinách, které když byly zobrazeny, vykazovaly vlastnost sebepodobnosti (self-similarity). Jak je vidět na Obrázku 1, Pokud se měnilo měřítko zobrazení množiny, její vzhled zůstával stále velmi podobný a objevovaly se stále se opakující vzory. Další důležitou vlastností byl velmi komplexní tvar, generovaný však sadou jednoduchých pravidel, které se aplikovaly stále dokola. B. Mandelbrot dal těmto objektům pojmenování fraktál. Slovo fraktál pochází z latinského slovesa fractus, které znamená zlomit, polámat a svým způsobem to velice dobře popisuje vzhled fraktálů [1].



Obrázek 1: Ukázka fraktálu „Mandelbrotova množina“ a jeho sebepodobnosti [2].

V přírodě se vyskytuje celá řada různých fraktálů. V roce 1904 vytvořil Helge von Koch fraktál vypadající jako sněhová vločka z rovnostranného trojúhelníku, viz Obrázek 3. Na jeho strany se opakovaně aplikovalo pravidlo, kdy se odstranila každá prostřední třetina strany a na jejím místě se vytvořil rovnostranný trojúhelník, sdílející právě tu chybějící část [3].



Obrázek 2: Fraktál „sněhová vločka“ je tvořen opakovaným aplikováním jednoho pravidla [3].

Dalších fraktálů, jež lze v přírodě nalézt je nespočet. Pokud začneme na lidském těle, fraktálem je například soustava krevního řečiště, jenž se větví z větších cév na menší a menší. Tím se stále zachovává vlastnost fraktálů – sebepodobnost a generování jednoduchými pravidly. Při přiblížení či oddálení vypadá síť cév stále velmi podobně. Jinými fraktály mohou být také lidské plíce, ledviny nebo nervová soustava. Mimo lidské tělo to může být vodní řečiště, tolik podobné tomu krevnímu, pobřežní linie, horská pohoří, vodní víry, bouřkové systémy, celé galaxie, ale i rostliny, stromy, listy a nespočetné spousty dalších [4,5].

Jak již bylo zmíněno výše, fraktály jsou i pohoří, pobřeží nebo vodní řečiště. Z této myšlenky vychází i většina algoritmů určených ke generování terénu. Není možné, aby se opakovaným aplikováním několika pravidel dokázal vytvořit komplexní a realisticky vypadající terén?

V roce 1982 se to stalo skutečností, když vyšel první film na světě, který obsahoval scénu s planetou, jejíž povrch byl kompletně generován počítači za použití fraktálové geometrie [7] – Star Trek II: Khanův hněv [6].

2.2 Přístup ke generování terénu

Procedurální generovaní mapy je možné rozdělit na několik částí, jelikož generovaná mapa není tvořena pouze terénem, ale skládá se z celé řady různých úrovní prvků mapy. Tyto úrovně na sebe navazují a postupně se skládají dohromady. Prvním a hlavním úkolem je vytvoření výškové mapy terénu, která je základem každé mapy.

Výšková mapa sama o sobě nenese informaci o typu terénu ani o jeho vlastnostech, ale pouze informaci o výšce v určitém bodě. Terén mapy se vytváří až interpretací této hodnoty. Ta se může v různých případech lišit a z jedné výškové mapy je tak možné vytvořit větší množství rozdílných map terénu.

Další úrovně prvků se generují do již vytvořeného terénu. Ty se mohou lišit na základě specifických potřeba dané aplikace. Typicky se ale jedná o vodní toky nebo lesy. Kromě přírodních prvků lze vygenerovat do výsledné mapy také herní prvky, kterými můžou být celá města, či strategické suroviny. O generování těchto úrovní se lze dozvědět více v kapitole 2.6.

K procedurálnímu generování výškové mapy terénu slouží celá řada algoritmů. Nejběžnějším přístupem je vytvoření šumu, se kterým lze dále pracovat jako s výškovou mapou. Nicméně vytvořený šum je použitelný i k jiným účelům, jako je třeba simulování vzhledu kouře, či oblak [8]. Mezi takovéto algoritmy se řadí například *Perlin noise* nebo *Diamond-Square Algorithm*. Jiné algoritmy již nejsou natolik obecné a jsou určené především ke generování výškové mapy. Takovým algoritmem je třeba *Hill Algorithm*, který produkuje velice specifické výsledky, typické svými zaoblenými tvary. Ty mohou být v některých případech žádoucí, neboť takto vygenerovaná výšková mapa terénu působí pohádkovějším dojmem. Existují také speciální algoritmy, jakými jsou třeba *genetické algoritmy*. Ty jsou určené k řešení obecných problémů, při kterých není dán přesný výsledek, ale stačí pouze aproximované řešení. Jejich použití ke generování terénu je ovšem komplikovanější. Zvláště z toho důvodu, že je potřeba vygenerovaný terén matematicky popsat a ohodnotit jeho vhodnost. Každý z těchto algoritmů má odlišné vlastnosti a generuje více či méně rozdílné výsledky.

Základním kamenem je tedy vytvoření samotné výškové mapy terénu. Zde je podrobnější přehled některých algoritmů sloužících k tomuto účelu.

2.2.1 Perlin noise

Perlin noise je metoda pro generování šumu, kterou vymyslel profesor Ken Perlin z New York University [9]. Existují dvě verze implementace tohoto algoritmu. První metoda je založena na vytvoření několika vrstev šumu s různou frekvencí a následném průměrování těchto vrstev. Nejprve je vygenerováno pole náhodných čísel. Z těch se vytváří takzvané *oktávy*. *Oktáva* je jedna vrstva, která vzniká vybráním k-tých hodnot pro k-tou vrstvu a následnou interpolací hodnot mezi nimi. Tím vzniknou šumy s různou frekvencí a tedy i různou úrovní detailů. Tyto vrstvy se poté průměrují dohromady a vytvoří výsledný šum, který je díky různým frekvencím oktáv soběpodobný na různých úrovních přiblížení.

Druhou verzí implementace je vytvoření pole gradientů. Toto pole má několikanásobně méně hodnot než výsledné. Je to pouze jakási síť. Každý bod, jehož hodnotu chceme získat, se nachází ve čtveřici bodů této sítě. Pro ty body, které chceme spočítat, se pak vypočítávají vektory z nejbližších čtyř bodů sítě gradientů. Skalárním součinem těchto vektorů s gradientem sítě v bodě, ze kterého vektor vychází, získáme vliv jeho hodnoty na výsledný bod, jehož hodnotu chceme zjistit. Pokud zprůměrujeme hodnoty všech čtyř nejbližších bodů, jejichž vliv jsme spočítali, získáme hodnotu šumu v konkrétním bodě.

2.2.2 Simplex noise

Tato metoda je vylepšenou verzí *Perlin noise* algoritmu. Odstraňuje některé artefakty, které mohly vzniknout, a také snižuje výpočetní složitost. Toho dosahuje počítáním s takzvanými *simplexy*. *Simplex* je n-rozměrné zobecnění trojúhelníku [10]. Oproti *Perlin noise* algoritmu snižuje výpočetní nároky díky výpočtům pouze s nejbližšími třemi body sítě gradientů (v případě 2D šumu) místo čtyř v *Perlin noise* algoritmu. Tato jeho přednost se projeví zejména ve vyšších dimenzích, kdy jeho výpočetní náročnost $O(n^2)$ pro n dimenzí mnohonásobně předčí výpočetní náročnost *Perlin noise* algoritmu $O(2^n)$ [11].

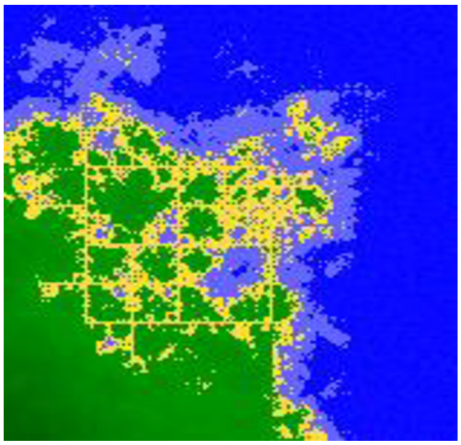
2.2.3 Algoritmus Midpoint-Displacement

Princip algoritmu spočívá v rozdělování generované oblasti na čtyři menší a náhodném upravování hodnot nově vložených bodů dělicích menší oblasti od sebe. To je následně rekurzivně opakováno pro každou novou oblast.

Algoritmus je určen pro výsledné čtvercové pole hodnot. Na začátku jsou vygenerovány čtyři první hodnoty, v každém rohu generované plochy jeden. Poté se vypočítá hodnota bodu uprostřed každé čtveřice bodů tvořící čtverec, a to tím způsobem, že se zprůměrují hodnoty rohů čtverce a přidá se k nim náhodná odchylka. Podobně se pokračuje i pro každou stranu těchto čtveřic, či čtverců. Bod uprostřed strany se získá zprůměrováním jejích krajních bodů a přidáním náhodné odchylky. Takto vznikne nová mřížka, která obsahuje čtyřikrát více menších čtverců. Tento postup se pak opakuje pro každý nově vzniklý čtverec, dokud není dosaženo požadovaného množství vygenerovaných bodů.

2.2.4 Algoritmus Diamond-Square

Jedná se o vylepšenou verzi *midpoint-displacement* algoritmu. *Diamond-square* algoritmus se liší výpočtem bodů uprostřed stran jednotlivých čtverců. Hodnota nového bodu na hraně již nezávisí pouze na dvou krajních bodech, ale i na bodech, jež jsou středy čtverců sousedících s hranou. Spojením těchto čtyř bodů vzniká tvar diamantu, což dalo tomuto algoritmu název.



Obrázek 3: Čtvercovité artefakty u midpoint-displacement algoritmu

Tímto přístupem se eliminují některé artefakty, které mohou vzniknout u *midpoint-displacement* algoritmu, a které jsou vidět na Obrázku 3. U toho se objevují především čtvercovité artefakty vznikající průměrováním hodnot pouze v horizontální, či vertikální rovině, neboť se hodnota uprostřed strany čtverce dopočítávala pouze ze dvou krajních bodů ležících v přímce a nebrala ohled na okolí.

2.2.5 Genetické algoritmy

Dalším, i když ne již tak běžným přístupem ke generování terénu, jsou genetické algoritmy. Genetické algoritmy jsou schopné řešit celou řadu obecných problému co nejlepší aproximací jejich řešení. Jsou inspirovány přírodou a fungováním křížení a přirozené selekce. Genetické algoritmy napodobují tuto situaci tím, že vygenerují celou sadu náhodných řešení. Každému řešení se říká *genom*. Tato řešení se poté spolu „kříží“. Vymění si část svého řešení mezi sebou a vzniknou dvě nová řešení, vzniklá kombinací jejich „předchůdců“. Kromě pouhého křížení výsledků se přidávají s jistou mírou i náhodné *mutace*, které mohou přinést další změny řešení. Aby se dosáhlo zlepšování výsledků, tak podobně jako v přírodě, mají „silnější“ větší šanci se rošířit. U genetických algoritmů se tento princip projevuje tím, že každé řešení musí být ohodnoceno nakolik je vhodné, a poté mají řešení s nejlepším ohodnocením větší šanci na „zkřížení“. Při každé iteraci se tedy kombinují nejlepší řešení s jinými nejlepšími řešeními. To se opakuje, dokud nemá mapa dostatečně dobré ohodnocení, nebo se nedosáhne limitu iterací.

Tento přístup má při generování map velkou nevýhodu. Kromě toho, že terén musí být matematicky popsaný, musí také existovat funkce, která by dokázala zhodnotit kvalitu daného řešení, což je velmi obtížné. Nicméně to není nemožné a existují i takové algoritmy, o kterých je možné se dočíst například zde [12].

2.2.6 Worley noise

Tato metoda vytváří specifický vzhled šumu, který se velmi podobá vzhledu buněk v mikroskopu. Algoritmus funguje na principu, kdy nejdříve vygeneruje náhodné body v cílovém prostoru. Hodnota šumu v konkrétním bodě se pak vypočítá jako vzdálenost k nejbližšímu z těchto bodů. Existují různé variace tohoto algoritmu, kdy se počítá vzdálenost jiného, než nejbližšího bodu, či se kombinují vzdálenosti několika bodů. Tento algoritmus je však náročnější použít ke generování terénu, neboť k dosažení rozumných výsledků je potřeba experimentovat s různými druhy výpočtů hodnoty šumu. Tyto výpočty lze navíc různě kombinovat. Právě tím, je ale tento algoritmus možný významnou měrou upravovat a vyladit na požadovaný výsledek.

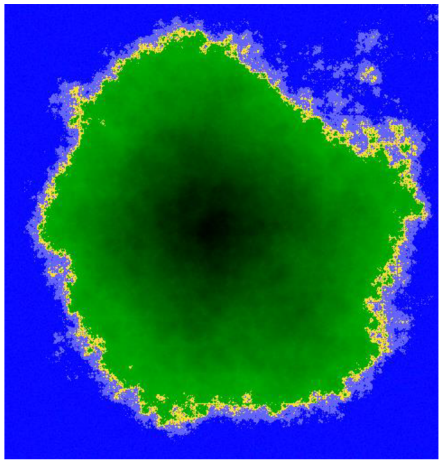
2.2.7 Hill algorithm

Hill algorithm, jak již sám název napovídá, je založen na nahodném generování „kopců“ do cílové plochy. Na počátku je plocha rovná. Do této plochy se zvolí náhodný bod, který bude středem kopce. Navíc se k tomuto bodu zvolí i náhodný poloměr a výška kopce v předem zadaných mezích. Výška tohoto nově vygenerovaného kopce nad původní rovinou se v každém bodě v tomto okruhu přičte k již existující výšce povrchu. To se opakuje stále dokola, takže kopce se různě překrývají a ovlivňují.

2.3 Diamond-square algoritmus

V tuto chvíli je důležité se rozhodnout, za jakým účelem má být daný terén generován, a který z algoritmů je vhodné použít. Tato práce má za cíl vytvořit mapu pro strategickou hru. Hlavním účelem vygenerovaného terénu je jeho funkcionalita a použitelnost pro danou aplikaci. Z tohoto důvodu, je vhodné, či v závislosti na konkrétních požadavcích na strategickou hru přímo nezbytné, mít možnost uměle zasáhnout do tvorby terénu a vnutit mu některé požadované prvky. Například by nebylo žádoucí, kdyby se jeden z uživatelů ocitl na mořském dně, protože by se pro něj zrovna nevygeneroval žádný ostrov.

Z dostupných algoritmů byl vybrán *diamond-square* algoritmus. Nejenom, že dosahuje velmi dobrých výsledků v oblastech reálného vzhledu a komplexnosti vygenerovaného terénu, ale hlavně, lze mu předdefinovat dopředu některé hodnoty. Jak je vidět na Obrázku 4, lze tak ovlivnit vzhled a chování celých oblastí, což byla vlastnost, která byla od vybraného algoritmu požadována.



Obrázek 4: Tvar ostrova byl algoritmu diamond-square předdefinován dopředu.

S prvním nápadem přišli již v roce 1982 pánové A. Fournier, D. Fussel a L. Carpenter [13]. Jejich algoritmus však vykazoval značné vertikální i horizontální artefakty v důsledku toho, že se výpočty pohybovaly v pravoúhlé mřížce [14]. Tento efekt byl potlačen vložením dalšího kroku do algoritmu, který počítá nové hodnoty pod úhlem 45° oproti ostatním výpočtům.

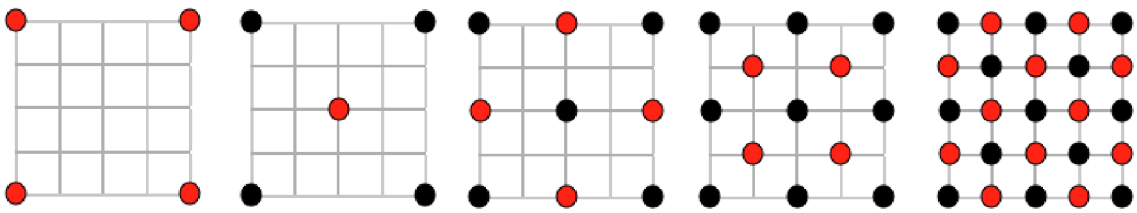
Tento algoritmus se využívá k vytváření šumu a to především v jednom a ve dvou rozměrech. Zároveň lze díky jeho implementaci zásadně ovlivňovat konečný výsledek upravováním různých parametrů. Toto je jeden z důležitých faktorů, protože máme k dispozici způsob pro vytváření právě takových map, které mají požadované vlastnosti.

Celá myšlenka algoritmu se nese v duchu fraktálů. Prostor je opakovaně dělen na podprostory a v těch jsou dopočítávány nové hodnoty. Algoritmus pracuje se čtvercovým prostorem, ale po jistých úpravách ho lze použít i v některých prostorech obdélníkového tvaru. V těch však musí být vygenerováno dostatečné množství bodů, které vytvoří pravoúhlou mřížku tvořící čtverce.

Pro tento algoritmus je nejprve potřeba vygenerovat první body, které jsou poté využity k dalším výpočtům. Těmto bodům přiřadíme náhodně vygenerované hodnoty v požadovaném rozsahu, a každý z nich umístíme do jednoho rohu mapy.

Algoritmus dále pokračuje dvěma kroky, které se opakují stále dokola. Prvním z nich je tzv. *diamond step* – diamantový krok. Tento krok pro každý ze čtverců (zezačátku existuje pouze jeden velký) vypočítá hodnotu v jeho středu. Ta je určena jako průměr z hodnot rohů čtverce sečtené s náhodně vygenerovanou hodnotou, která vnáší prvky náhodnosti terénu. Rozsah této hodnoty je možné přizpůsobovat a měnit, což je jeden z mechanismů ovlivňující vygenerovaný terén.

Dalším krokem je tzv. *square step*, který se provádí pro všechny *diamanty* – čtverce otočené o 45°. U tohoto kroku je nutné řešit situaci na krajích, kde chybí některé hodnoty. Způsobů řešení existuje několik. Hodnoty je možné náhodně vygenerovat, lze se spokojit pouze se 3 hodnotami nebo je možné použít hodnotu na protější straně mapy. Nově vygenerované body po obou krocích rozdělují výslednou plochu na čtyři podprostory, jak je možné vidět na Obrázku 5. Na každý z těchto podprostorů je poté použit stejný postup, kterým jsou opět rozděleny na čtyři nové podprostory. To se opakuje do chvíle, kdy je vygenerováno požadované množství bodů.



Obrázek 5: Ukázka jednotlivých kroků algoritmu diamond-square [15].

Tento algoritmu má několik výhod. Prvně je to možnost zasahovat do vzhledu výsledného terénu. Toho lze dosáhnout vložením požadované hodnoty na pozici, která ještě nebyla vygenerována. Když algoritmus pozná, že se jedná o jinou než počáteční hodnotu, tak tuto pozici nebude přepisovat a ponechá zde vloženou hodnotu. Vložená hodnota je pak používána při výpočtech dalších oblastí a tím ovlivňuje celé své okolí. Musí se však vzít v potaz, kdy se bude s danou pozicí počítat. Čím později bude použita, tím je její vliv na okolí menší. Další výhodou je rychlost algoritmu – výpočty jsou méně náročné, jelikož se jedná především pouze o průměrování hodnot.

Asi největší nevýhodou *diamond-square* algoritmu jsou přesně dané rozměry. Výška i šířka se u mapy se standartním čtvercovým tvarem řídí vztahem:

$$s = 2^x + 1 \qquad (1)$$

Kde *s* značí počet bodů terénu na jedné straně mapy a *x* je kladné celé číslo, značící počet iterací algoritmu.

2.4 Regulace generovaného terénu

Cílem ovlivňování výsledného reliéfu terénu nejsou zásahy na nižších úrovních, jakými jsou třeba zakřivení pobřežní linie, ale spíše na celkové logice vygenerovaného terénu. Smyslem je pouze zadat si požadavky či parametry vygenerovaného terénu a nechat algoritmus, ať se postará o zbytek. Jedním z těchto požadavků je celková geometrie vygenerovaného terénu. Ten sice může být vytvořen zcela náhodně, ale často je potřeba zajistit některé základní potřeby. Z pohledu strategických her se může jednat o výchozí pozice hráčů. U zcela náhodné mapy nelze žádné takové vhodné místo zaručit, a proto je mimojiné potřeba mít možnost zasáhnout do výsledného terénu.

Další pohnutkou můžou být různé strategické vlivy terénu. Těmi lze rozumět například umělé oddělení částí pro jednotlivé hráče. Nemožnost dostat se k protihráči v počátečních fázích hry, dokud nelze oddělení překonat, může dokonce zcela změnit běh hry a vnést nové oživující prvky. Toho lze také docílit vhodně použitou vodní plochou, která může ve hře jednak sloužit specifickým účelům a novým herním principům, ale i výrazně oživuje vzhled celého povrchu a přidává mu na přitažlivosti.

Kromě toho, že určíme některé základní požadavky a algoritmu předdefinujeme námi požadované prvky, je možné výsledný terén ještě dále ovlivňovat. Především lze regulovat hrubost vygenerovaného terénu, a to na několika úrovních. *Diamond-square* algoritmus začíná generovat pouze z několika málo počátečních bodů, ale tyto body mají velký vliv na celý vygenerovaný terén. Čím více se liší hodnota výšky těchto bodů, tím více se promítnou výškové rozdíly i do celého terénu. Naopak body s podobnou výškou vytvoří terén velmi rovinatý. Kromě těchto počátečních bodů se přidává ke všem nově vygenerovaným bodům náhodná hodnota, aby byla do generovaného terénu vnesena jistá nepravidelnost. Tato náhodná hodnota se zmenšuje s tím, čím menší oblast ovlivňuje, jinak by v extrémním případě docházelo k tomu, že by dvě sousední místa byla příliš rozdílná a mapa by byla plná vysokých a uzkých „komínů“ a propadlých „děr“. Ovlivňováním toho, jak rychle se tato náhodná hodnota zmenšuje, je další z přístupů, kterými lze regulovat hrubost generovaného terénu.

2.5 Dotváření povrchu terénu

Máme-li vygenerovanou výškovou mapu, je potřeba ji nějakým způsobem interpretovat, získat z ní mapu terénu pouze ve dvou rozměrech. Základním a intuitivním přístupem je získání typu terénu podle výšky. Toho lze docílit rozdělením celkové výšky na několik úrovní. Každá z těchto úrovní pak může být znázorněna různými druhy povrchu terénu. Každá úroveň následně odpovídá jinému typu terénu.

To může vypadat tak, že za nejnižší úroveň jsou označeny všechny body, jež spadají do spodních 10% celého výškového rozsahu. Této úrovni je pak přiřazen druh terénu. Tím může být třeba vodní hladina. Poté, u každé hodnoty ve výškové mapě, která spadá do této nejnižší úrovně, můžeme označit jí odpovídající pozici v mapě terénu jako terén typu vodní hladina. Analogicky

můžeme postupovat u dalších úrovní. Použitím rozdílných typů terénu a úrovní lze dosáhnout značné škály zcela rozdílných map.

Příkladem může být vytvoření terénu použitelného pro mapy do oblastí s horkým podnebím. Ten lze například získat označením nízko položených míst terénem močálu místo vodní hladiny, výše položeným místům přiřadit pouštní terén a zbylý prostor doplnit pralesem. Aby nebyly přechody mezi těmito úrovněmi příliš násilné, lze mezi ně vložit uzké pásma trávy, nebo jiného terénu, jenž zjemní přechod. Obdobně je možné okolo vodních ploch přidat pobřežní pásma.

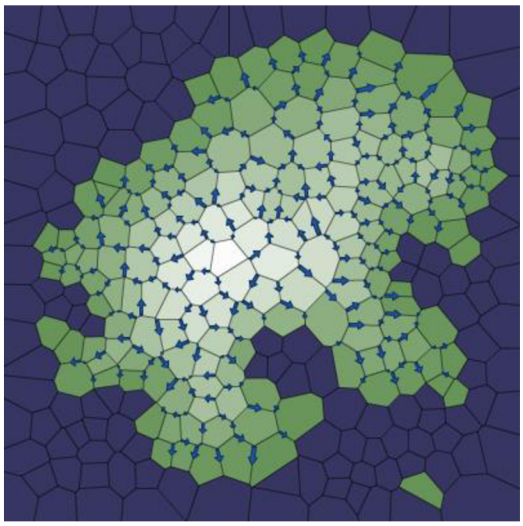
Určování terénu nemusí probíhat pouze z výškové mapy, ale lze využít i jiné principy. Takovým může být prostorová lokalita, kdy kromě výšky v daném bodě ovlivňuje výsledný povrch terénu i jeho pozice na mapě. V případě rozlehlejších map tak lze simulovat různé podnebné pásy.

Efektu, kdy horní část mapy reprezentuje arktickou oblast a směrem ke spodnímu okraji mapy se terén mění přes mírný podnebný pás až k tropickému podnebnému pásu, lze dosáhnout rozdělením mapy na horizontální pásy, v nichž se aplikují různé druhy povrchu terénu i na pozicích spadajících do stejné výškové úrovně. Na arktické části mapy tak může být vystřídán travnatý terén zamrzlou půdou, i když oba druhy povrchu spadají do stejné výškové úrovně.

2.6 Generování dalších úrovní mapy

Přestože získaná mapa terénu patří z hlediska strategických her k tomu nejdůležitějšímu, samotná působí pustým dojmem. Nejenom ke zvýšení její atraktivity je vhodné do vytvořeného terénu přidat i další prvky. Především se jedná o krajinné prvky, jako jsou vodní toky a lesy. Ty lze zapracovat do herních principů, například, když je pro některé stavby potřeba blízkost vodního toku nebo stromů, ale také mohou plnit pouze esteticku funkci. Generování těchto prvků proto úzce souvisí s konkrétní aplikací a jejich účelem a způsobem, jakým mají být do hry integrovány.

Rozhodneme-li se o přidání vodních toků, je důležité, jestli jejich výskyt ovlivňuje herní principy. Pokud ovlivňuje, je potřeba ohlídat, aby některý z uživatelů nebyl znevýhodněn jejich špatnou polohou. Způsob generování se pak musí odvíjet především s ohledem na jejich strategický vliv. Dalším prvkem, který hraje velkou roli, je samotné zobrazení mapy. Pokud se jedná o trojrozměrnou mapu, nelze vodní tok generovat zcela náhodně, ale je vhodné zachovávat zákony fyziky. Vodní tok se tak musí pohybovat vždy směrem nejmenšího odporu, čímž je místo, kde je spád terénu největší, viz Obrázek 6.



Obrázek 6: Vizualizace velikosti sklonu terénu [16].

Samotné generování může simulovat přirozené chování vody v přírodě. K tomu je nejdříve potřeba spočítat z výškové mapy mapu gradientů terénu. Ty nám dávají informace o sklonu terénu, jak je vidět na Obrázku 6. Poté může být vybráno náhodné místo, nejlépe výše položené, a označeno za pramen. Z této pozice se pak vždy, za pomoci mapy gradientů, vede vodní tok tím směrem, kde je největší sklon terénu [16].

Pokud terén žádným směrem neklesá, může se v tomto místě začít tvořit jezero. Pokud se jeho hladina dostatečně zvedne na úroveň, kde již existuje spád a jezero „přeteče“, pokračuje vodní tok dál, dokud se nedostane na ukončující místo, kterým může být úroveň moře. Druhou možností je, že jezero přestane stoupat nad učitou úroveň a stane se samo ukončujícím místem vodního toku.

U dvojrozměrných map je situace trochu jiná. Z důvodu, že na nich v rámci jedné výškové úrovně není vidět jejich sklon, není nutné určovat směr vodního toku pouze na základě gradientu terénu, ale lze si ho přizpůsobovat, případně ho lze vygenerovat zcela nezávisle za použití primitivnějších metod. Takovou metodou je například nalezení nejbližšího ukončujícího bodu a jeho spojením s počátkem nejkratší nalezenou cestou, která však nesmí přejít z nižší úrovně do vyšší. Jelikož vodní tok málokdy teče přímo, může být jeho tvar upraven dodatečnými zásahy. Vodní tok lze v různých místech vybočit a opět přivést zpět a tím simulovat meandrovité chování [17].

Podobná situace nastává i při generování stromů a lesů. Velkým faktorem je zde jejich účel. V některých hrách jsou lesy monotónní, v jiných se jedná spíše o samostatné stromy a jejich shluky a v některých se téměř nevyskytují. Lesy také mohou mít strategickou funkci, kdy fungují jako neprůchodné, nebo pouze částečně průchodné oblasti. Taktéž mohou sloužit jako strategická surovina, která je těžena. Pokud lesy slouží jako strategická surovina, prioritou při jejich generování je rovnoměrné a vyvážené rozložení přes všechny hráče, aby nebyl nikdo znevýhodněn.

Nejsou-li kladeny při generování stromů specifické podmínky na herní principy, řídí se většina přístupů, podobně jako u řady procedurálně generovaných věcí, paralelami s přírodou. Základem je vytvoření několika druhů stromů, kdy každý z nich má své vlastnosti, co se týče jejich výskytu a hojnosti. Například, jehličnaté stromy budou růst pravděpodobněji ve vyšších, či chladnějších oblastech, palmy naopak v teplých a vlhkých oblastech. Stromy pak lze vygenerovat náhodně, s přihlednutím k tomu, kde se který typ stromu a s jakou pravděpodobností vyskytuje. Množství vygenerovaných stromů také může být spojen s výškovou úrovní i s klimatickými podmínkami dané mapy.

Takto vygenerované stromy mohou být v některých případech dostačující, ale jejich náhodné generování nereflektuje dostatečně vzhled skutečných lesů. V těch platí určitá pravidla, kdy stromy nerostou náhodně, ale zohledňují celou řadu faktorů, jako je poloha okolních stromů, druhy okolních stromů či klimatické podmínky. Jedním ze způsobů, jak zohlednit tyto vlivy, je zavedením ochrané zóny okolo každého stromu, ve které nesmí existovat žádný jiný strom, neboť pokud rostou dva stromy příliš blízko sebe, pravděpodobně jeden z nich uhyne z nedostatku živin nebo kvůli jinému k růstu potřebnému faktoru. Jiným přístupem pak je vygenerování pravidelných rozestupů mezi stromy a jejich následném posouvání náhodným směrem tak, aby se nekryly s jinými stromy [18].

Pro dosažení reálnějšího vzhledu lesů se používají simulace systémů lesního porostu [19]. Tyto modely zohledňují reálné vlastnosti stromů, jakými jsou například šíření jejich semen, rychlost růstu, odolnost, vhodnost daného prostředí a spousty jiných. Zpočátku se „vysadí“ několik prvních exemplářů stromů a simuluje se jejich růst. Každý odrostlý strom pak vypouští svá semena a umožňuje růst nových stromů. Ty mohou odumřít nebo prosperovat v závislostí na okolních podmínkách. Po simulaci několika životních cyklů se vygenerovaný les použije jako výsledný. Tato metoda sice vykazuje nejreálnější výsledky, je však také náročnější na implementaci.

3 Hledání cesty v mapě

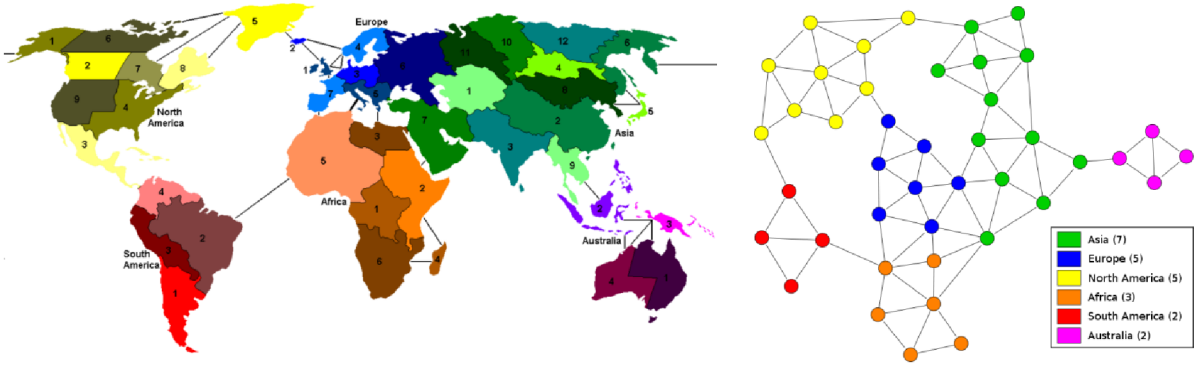
3.1 Reprezentace problému

I když si to nemusíme uvědomovat, setkáváme se s hledáním nejkratší cesty prakticky každý den, ať už jde o GPS navigaci, prohlížení internetových stránek, kdy je vyhledávána cesta paketům přes síť, nebo vyhledávání návaznosti spojů MHD. Obecněji jde o hledání nejkratšího či nejlepšího postupu k dosažení cíle, neboť nejkratší postup může být jen jedním z mnoha kritérií popisujících nejlepší řešení.

Aby bylo možné algoritmicky řešit všechny tyto případy, je nejdříve potřebné správně daný problém reprezentovat. Vhodnou reprezentací problému a dat potřebných k jeho vyřešení se nám dostává do ruky mocný nástroj, který umožňuje daný problém efektivně řešit.

V případě hledání cesty v mapě je potřeba reprezentovat každý bod, na který je možný se v mapě pohnout. Také je potřeba znát vztahy mezi body, aby bylo možné určit, které spolu sousedí a mezi kterými lze přecházet. V případě, kdy různý terén má různé vlastnosti a přechod mezi jednotlivými body na tomto terénu se liší od jiného, je potřeba znát i tuto informaci.

Pro tyto požadavky je vhodná reprezentace pomocí ohodnoceného grafu. Vezmeme-li v úvahu mapu, lze všechny schůdné body reprezentovat uzly v grafu a přechodu mezi nimi a jeho cenu jako ohodnocenou hranu. Toho si je možné všimnout na Obrázku 7. Převedeme-li mapu do grafové reprezentace, lze na ni již použít klasické algoritmy určené k hledání nejlépe ohodnocené cesty.



Obrázek 7: Mapa ze stolní hry Risk [20] a grafová reprezentace této mapy [21].

3.2 Metody prohledávání grafů

K nalezení cesty se využívá grafové reprezentace prohledávaného prostoru a tudíž lze použít algoritmy, které jsou zaměřeny na hledání nejlepší cesty v grafech. Tyto metody se dělí podle toho, jestli mají informace o tom, kde se nachází hledané řešení. Metodám, které touto informací disponují a využívají ji se říká *informované* metody. Metodám bez této informace se říká *neinformované*.

3.2.1 Primitivní metody

Primitivním přístupem k nalezení cesty je prohledávání prostoru směrem k cíli. Při střetnutí se s překážkou se daná překážka obchází, a jak je to jen možné, opět se prohledává směrem k cíli. Tento způsob je neefektivní a navíc ani nemusí najít řešení, a proto se nevyužívá.

3.2.2 Breadth First Search

Jednoduchý algoritmus, který postupně prochází celý prostor. Z každého uzlu, který prohledává, vždy přidá všechny sousední uzly k dalšímu prohledávání. Tento postup opakuje stále dokola tak dlouho, dokud mu nedojdou další uzly k prozkoumávání, nebo nenarazí do cíle.

Tento algoritmus se řádí mezi neinformované metody, jelikož nezohledňuje polohu řešení, které se snaží nalézt. Algoritmus je úplný i optimální pro konečné množství uzlů. „Je-li počet bezprostředních následníků každého uzlu konečný, pak algoritmus BFS je úplný a optimální. Časová i paměťová náročnost algoritmu BFS je však exponenciální – je dána výrazem $O(b^{d+1})$, kde b je tzv. faktor větvení (*branching factor*), tj. průměrný počet bezprostředních následníků každého uzlu, a d (*depth*) je hloubka nejlepšího řešení, tj. řešení, které se nachází v nejmělčí hloubce.“ [22 s.16]

3.2.3 Dijkstrův algoritmus

Tento algoritmus funguje podobným způsobem jako algoritmus *Breadth First Search*. Rozdílem je, že podporuje i ohodnocené hrany. Díky tomu je tento algoritmus schopen vzít v úvahu i ohodnocené hrany, kde se liší cena přechodu. Nový uzel k rozgenerování nevybíra podle toho, jestli je první ve frontě, ale podle toho, který z uzlů určených k rozgenerování má nejmenší hodnotu cesty od počátku.

U každého z nově rozgenerovaných uzlů se nejdříve zjistí, jestli už nebyl do seznamu uzlů k rozgenerování přídán v nějakém jiném kroku a pokud byl, porovnává se, není-li hodnota cesty přes aktuální uzel do tohoto uzlu menší, než je předchozí uložená hodnota cesty. Na začátku mají všechny nenavštívené uzly hodnotu „nekonečno“, tudíž je cesta do nich vždy kratší.

Algoritmus se řadí mezi neinformované metody, jelikož nezohledňuje polohu řešení, které se snaží nalézt. Algoritmus je úplný i optimální s časovou náročností $O(n^2)$ [23], kde n značí počet uzů, při použití seznamu či pole pro uložení hodnot. Při použití Fibonacciho haldy se časová náročnost zmenší na $O(n \log n + m)$ [24], kde n značí počet vrcholů a m počet hran.

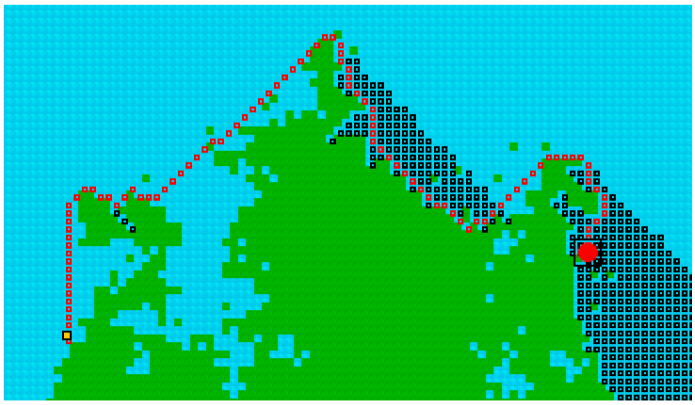
3.2.4 Best First Search algoritmy

Best First Search je skupina algoritmů, patřících mezi informované metody, které jsou založené na tom, že rozgenerovávájí nejlépe ohodnocený uzel jako první. Ohodnocení každého uzlu je dáno vztahem:

$$F(n) = G(n) + H(n) \qquad (2)$$

Kde n je uzel, pro který se počítá jeho hodnota, $G(n)$ je hodnota nejkratší nalezené cesty od počátku do tohoto uzlu a $H(n)$ je heuristický odhad ceny cesty do cíle. Podle této hodnoty jsou vybírány další uzly k rozgenerovávání. Tímto způsobem, kdy je v hodnotě uzlu započítán i odhad ceny cesty do cíle, je umožněno vynechat prohledávání uzlů, které jsou potenciálně příliš vzdálené od řešení problému a zaměřit hledání směrem očekávaného řešení.

Greedy search – Varianta *Best First Search* algoritmu, který bere v potaz k ohodnocení uzlu pouze heuristickou složku $H(n)$. Tato varianta algoritmu je typická tím, že se snaží dostat co nejrychleji co nejblíže k cíli. U tohoto přístupu však může dojít k problému, kdy dostanou přednost ty uzly, které se zdají být blíže cíli, ovšem nevede přes ně nejlepší cesta. Jak si lze všimnout na Obrázku 8, algoritmus se snaží dostat na co nejkratší vzdálenost od cíle a z důvodu, že cena již nalezené cesty není započítána a tudíž nehraje roli, je algoritmus nakonec schopný celou cestu značně prodloužit.



Obrázek 8: Viditelná neoptimálnost cesty nalezené algoritmem greedy search. Nalezená cesta je značena červeně a prohledávané uzly černě.

A* algoritmus – Tato varianta *Best First Search algoritmu* využívá jak složky heuristické, tak složky skutečné ceny cesty do prohledávaného uzlu. Díky tomu upřednostňuje k rozgenerovávání ty uzly, které se zároveň blíží cíli a zároveň neprodloužují nalezenou cestu. Zde záleží chování algoritmu na vyvážení vlivu heuristiky $H(n)$ a ceny cesty $G(n)$.

Při splnění podmínky, že heuristická funkce je přípustná, což znamená , že odhadovaná cena cesty do cíle je menší než skutečná cena, je tento algoritmus úplný i optimální. „Časovou a prostorovou náročnost proto výrazně ovlivňuje použitá heuristika – pokud se blíží 0 (to je jistě spodní odhad skutečné ceny), pak složitost se blíží exponenciální složitosti, pokud je použitá heuristika dobrým spodním odhadem skutečné ceny, pak jsou expandovány pouze uzly kolem optimální cesty (je-li přesným odhadem, pak jsou expandovány pouze uzly na optimální cestě).“ [22 s.33]

3.3 Algoritmus A*

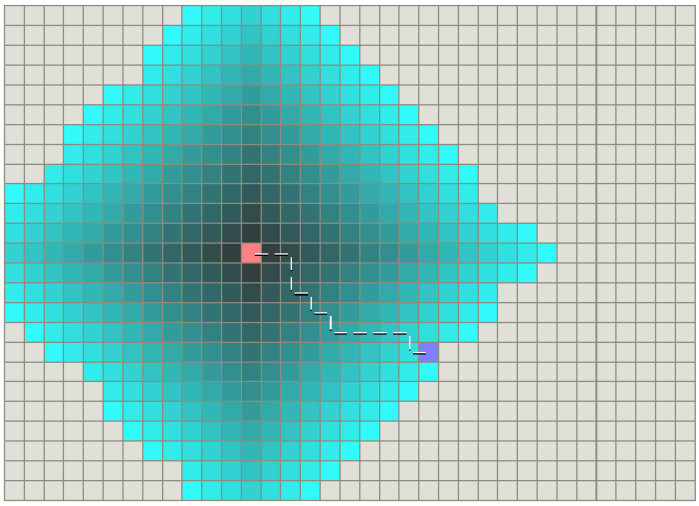
„Metoda A* … je nejznámnější a nejpoužívanějších metodou pro řešení úloh prohledáváním stavového prostoru.“ [22 s.33] Algoritmus A* kombinuje několik přístupů, ve kterých využívá výhod jiných metod. A* je taktéž algoritmus s vysokou mírou flexibility, jenž lze využít pro efektivnější řešení řady situací. Zároveň je schopný brát v potaz různé ohodnocení hran grafu, kterými lze simulovat různě schůdný terén. K tomu všemu, při správném použití heuristické funkce, garantuje nalezení nejkratší cesty.

Přístupy, které A* algoritmus kombinuje, jsou využití heuristiky (např. *Greedy Search*) a rozgenerovávání uzlu, který je nejblíže počátečního uzlu (např. *Dijkstrův algoritmus*).

Postup algoritmu Greedy Search je takový, že prohledává všechny sousedy uzlu s nejlepším ohodnocením heuristické funkce, která odhaduje cenu cesty do cíle. Algoritmus se tak snaží jít

nejkratší cestou k cíli. Dostane-li se do nevhodného místa, nalezená cesta může být značně nevýhodná, jak je vidět na Obrázku 8.

Oproti tomu, Dijkstrův algoritmus prohledává vždy z uzlu, jenž je neblíže počátku. Tím je garantováno nalezení nejkratší cesty. Na druhou stranu, algoritmus prohledává několikanásobně větší množství uzlů, jelikož nezohledňuje polohu cíle, což je zřejmé z Obrázku 9.



Obrázek 9: Dijkstrův algoritmus nezohledňuje polohu cíle a prohledává všemi směry. Prohledaný prostor je zvýrazněn tyrkysovou barvou [25].

U algoritmu A* je každý z uzlů ohodnocen hodnotící funkcí $F(n)$, která je dána jako $F(n) = H(n) + G(n)$, kde $H(n)$ je hodnota heuristiky v daném uzlu a $G(n)$ je cena cesty do daného uzlu. Uzly jsou tak hodnoceny podle své vzdálenosti od cíle, ale zároveň se dává přednost prohledávání k předpokládanému směru k cíli.

Díky tomuto způsobu ohodnocování uzlů je A* velice variabilní. Zde je uvedeno několik případů, které mohou nastat [26]:

- *V případě, že se bude H(n) rovnat nule, jedná se o Dijkstrův algoritmus.*
- *V případě, že by se G(n) rovnalo nule, algoritmus se bude chovat shodně s algoritmem Greedy Search.*
- *Pokud bude hodnota H(n) vždy nižší než skutečná cena do cíle, algoritmus zaručuje nalezení nejkratší cesty*
- *Přesahuje-li hodnota H(n) občas skutečnou cenu cesty do cíle, algoritmus nemusí vždy najít nejkratší cestu. Nalezená cesta jí však bude velmi podobná. Zároveň se sníží počet prohledávaných uzlů a tím zvýší rychlost.*

Hodnoty $H(n)$ a $G(n)$ musí být tedy vyrovnané, chceme-li, aby A* nacházel co nejlepší řešení, co nejrychleji. Máme zde ale možnost měnit poměr těchto dvou složek. Budeme-li snižovat hodnotu $H(n)$, nalezená cesta bude přesnější. Naopak, budeme-li zvedat hodnotu $H(n)$, snížíme tím i počet prohledáváných uzlů a tím zrychlíme celý algoritmus. To však za tu cenu, že nalezená cesta nemusí být nejkratší. Tímto způsobem je možné dynamicky ovlivňovat výkon.

3.3.1 Heuristické funkce

Cílem dobré heuristické funkce je co nejlépe odhadnout cenu nejkratší cesty k cíli z aktuálního uzlu. V ideálním případě, kdy by byla známa přesná cena z každého bodu k cíli, by A* nalezl ideální cestu, a přitom by neprozkoumával žádné uzly navíc, neboť by přesně věděl, který z okolních uzlů leží na optimální cestě.

Způsobů, jak co nejlépe odhadnout cenu cesty do cíle je několik. Pro zpřehlednění budeme značit rozdíl *x*-ové souřadnice cílového a aktuálního uzlu jako *dx*. Obdobně pro *y*-ové souřadnice to bude *dy*.

Manhattanská metrika – Odhad ceny nejkratší cesty jako součet *dx* a *dy*. Tato heuristika je velmi často využívána z důvodu její jednoduchosti a tedy i rychlosti výpočtu. Hodí se především na prostor, ve kterém se lze pohybovat pouze horizontálně a vertikálně. Lze však použít i na prostor umožňující i pohyb do úhlopříčky.

$$\mathrm{H(n)} = \mathrm{dx} + \mathrm{dy} \qquad (3)$$

Eukleidovská vzdálenost – Odhad ceny nejkratší cesty vypočítané jako přímá vzdálenost mezi počátkem a cílem. Tato heuristika je vhodná na prostor umožňující pohyb všemi směry. Oproti *Manhattanské metrice* je přesnější. Její nevýhodou je složitost výpočtu, která může způsobit zpomalení u prohledávání většího prostoru, kdy výrazně naroste počet uzlů, které je potřeba ohodnotit.

$$\mathrm{H(n)} = \sqrt{dx^2 + dy^2} \qquad (4)$$

Čebyševova vzdálenost – Odhad ceny nejkratší cesty jako maximum z *dx* a *dy*. Tato heuristika se používá v prostorech, kde je umožněn diagonální přechod a zároveň je ohodnocen stejně, jako přechod vertikální či horizontální. Toto chování ovšem nebývá u hledání nejkratší cesty v terénu žádoucí, jelikož diagonální přechod je „delší“ než horizontální, či vertikální.

$$\mathrm{H(n)} = \max(\mathrm{dx}, \mathrm{dy}) \qquad (5)$$

3.3.2 Optimálnost algoritmu A*

Algoritmus A* dokáže být velice variabilní a umí zkrátit své výpočty za cenu kvality nalezené cesty. Někdy je ale potřeba znát přesnou cestu. Algoritmus A* garantuje nalezení nejlépe ohodnocené cesty, a to za podmínky, že je použita přípustná heuristická funkce. Přípustná heuristická funkce musí vždy vracet menší nebo stejný odhad ceny cesty do cíle, než je jaká je její skutečná cena. Důkaz optimálnosti je převzat ze skript k předmětu základy umělé inteligence [22 s.33].

Pokud platí podmínka, že použitá heuristika je přípustná, lze dokázat, že algoritmus A nalezne optimální cestu:*

1. *Označme cenu optimální cesty do optimálního cílového uzlu jako* $f_0(s_{Gopt})$ *a cenu jiné (neoptimální) cesty do stejného nebo jiného cílového uzlu jako* $f(s_G)$*. Zřejmě musí platit* $f_0(s_{Gopt}) < f(s_G)$*.*

2. *Nechť x je uzel na optimální cestě k optimálnímu cíli. Pro přípustnou heuristiku musí platit* $f_0(s_{Gopt}) \geq f(x)$.
3. *Pokud by uzel x nebyl vybrán k expanzi, zatímco cílový uzel* s_G *s ohodnocením* $f(s_G)$ *ano, muselo by platit* $f(x) \geq f(s_G)$.
4. *Z bodů 2 a 3 vyplývá, že* $f_0(s_{Gopt}) \geq f(x) \geq f(s_G)$, *tedy že* $f_0(s_{Gopt}) \geq f(s_G)$, *což je ve sporu se závěrem bodu 1.*
5. *Uzel x proto musí být vybrán k expanzi, stejně jako každý jiný uzel na optimální cestě k optimálnímu cíli a metoda proto musí nalézt optimální cestu.*

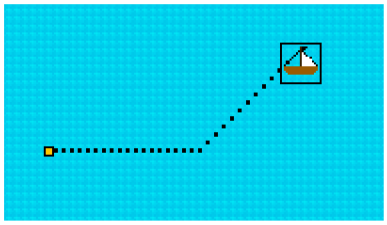
3.4 Víceúrovňové prohledávání prostoru a post-processing nalezené cesty

S nárustem velikosti mapy narůstá i množství prohledávaných uzlů. Při každém x-násobném zvětšení mapy se počet uzlů mapy zvětší x^2-krát. S postupným zvětšováním map počet uzlů velmi lehce dosáhne kritické hranice a rychlost celého algoritmu se prudce sníží.

To lze do jisté míry řešit rozdělením mapy na mapu terénu a na mapu herních objektů, kdy mapa herních objektů má větší velikost než mapa terénu. Díky tomuto přístupu je umožněno, aby se mohly herní objekty pohybovat volněji a nebyly fixovány na jedno políčko terénu. Místo toho, se mohou nacházet na několika různých místech, spadajících v mapě terénu pod jediný uzel. Mapa herních objektů by měla být násobkem velikosti mapy terénu, aby pod každý bod na mapě terénu spadal stejný počet bodů mapy herních objektů. Potom, označíme-li si tento poměr stran obou map jako R_m, obsahuje každé políčko terénu $(R_m)^2$ uzlů mapy herních objektů. Algoritmus pro hledání cesty pak prohledává pouze v mapě terénu, jelikož je jisté, že pokud je schůdný jeden bod v mapě herních objektů patřící pod jedno políčko terénu, tak jsou schůdné i všechny ostatní patřící pod to stejné políčko.

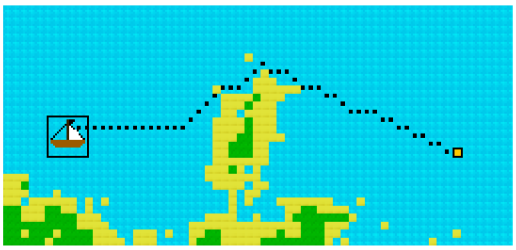
Výsledná cesta je nalezená pouze na úrovni mapy terénu a proto je potřeba ještě dohledat cestu mezi těmito nalezenými body i na úrovni mapy herních objektů. Nejjednodušším způsobem jak toho dosáhnout je jednoduchou interpolací chybějících bodů na herní mapě. Výsledná cesta vzniklá tímto postupem je téměř optimální, neboť nalezená cesta v mapě terénu optimální je, ale na „jemnější“ mapě herních objektů tato cesta již optimální být nemusí a to i z důvodu interpolace části bodů.

Daleko výraznějším prvkem je nerealistický vzhled některých nalezených cest, viditelný na Obrázku 10. Především se jedná o situace, kdy je umožněn pohyb osmi směry. Přestože nalezená cesta je optimální, nemusí kopírovat směr pohybu, a místo toho se snaží co nejdříve využít diagonálního pohybu k rychlejšímu přiblížení k cíli. Poté, co je některá ze souřadnic shodná s cílovým bodem, cesta již vede pouze vertikálně, či horizontálně.



Obrázek 10: Nalezená cesta je optimální, nevypadá však realisticky.

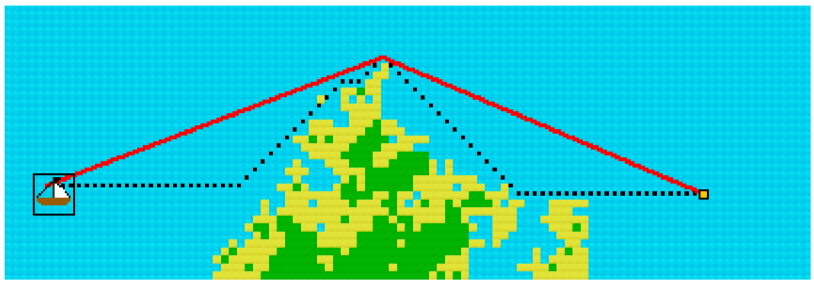
Toto chování lze v některých případech ovlivnit použitou heuristikou, která upřednostňuje uzly v přímém směru k uzlu cílovému. Takovou heuristikou je například *Eukleidovská vzdálenost*. Zásadní nevýhoda je ta, že není-li počáteční a cílový bod dostupný v přímém směru, ale cesta vybočuje, aby se vyhla překážce, heuristika stále upřednostňuje směr přímý k cílovému uzlu před směrem pohybu. Zlepšení nerealistického vzhledu cesty je touto metodou dosaženo pouze v místech, kde je momentální směr nalezené cesty shodný s přímým směrem k cílovému uzlu, jak je vidět na Obrázku 11.



Obrázek 11: Použití heuristiky *Eukleidovská vzdálenost* zlepší tvar cesty při přiblížení se k cílovému (oranžovému) bodu. Vlevo od překážky vypadá cesta stále nerealisticky.

Pro dosažení realističtěji vypadajících cest nebude stačit pouze upravovat algoritmus pro hledání cesty. Místo toho lze využít metody post-procesingu nalezené cesty, kterou postupně „vyhladíme“. Na Obrázku 12 lze porovnat tvar cesty před vyhlazením a po něm.

Výchozím bodem je jeden z konců nalezené cesty. Algoritmus pak postupně pokračuje k dalšímu bodu cesty a snaží se zjistit, jestli jde mezi těmito body nalézt přímou cestu. Pokud lze, vyzkouší další bod z původně nalezené cesty. Takto postupuje až do té doby, než narazí na uzel, ke kterému se již přímým směrem dostat nelze. Poté algoritmus uloží všechny body na přímce až k poslednímu bodu, ke kterému šlo najít přímou cestu, a nahradí jimi odpovídající část původní cesty. Postup algoritmu se opakuje vždy pro další úsek, navazující na naposledy nahrazenou část [27].



Obrázek 12: Porovnání nalezené cesty (černě) a vyhlazené cesty (červeně).

Velkou výhodou použití vyhlazování nalezené cesty je, že algoritmus je schopný pracovat i s body cesty nalezenými pouze v mapě terénu, avšak nově získané úseky umožňuje generovat na úrovni mapy herních objektů. Navíc je v tomto případě algoritmus zrychlen, neboť zkoumá cestu mezi několikanásobně menším množstvím uzlů.

3.5 Hierarchické algoritmy pro hledání cesty

Ne vždy stačí použití této dvouúrovňové architektury. S rostoucí velikostí prohledávaného prostoru již výše zmíněné postupy nejsou optimální a je potřeba pozměnit přístup. Možným řešením je využití hierarchických algoritmů pro hledání cesty [28]. Tyto algoritmy vycházejí z myšlenek podobných dvoúrovňové architektuře, ale posouvají je dál. Mapu již nedělí pouze na dvě úrovně, ale na libovolné vhodné množství úrovní. Cesta se pak postupně vyhledává od nejobecnější úrovně až k nejkonkrétnější úrovni. Ta odpovídá mapě herních objektů a určuje již přesné pozice k pohybu.
V každé úrovni je mapa rozdělena na stejně velké oblasti, takzvané *clustery*. Každý z těchto clusterů obsahuje několik hraničních bodů a informaci o tom, na který hraniční bod sousedního *clusteru* se z nich lze dostat. Také vzdálenosti mezi hraničními body uvnitř jedné oblasti jsou předpočítány.

Hledání cesty pak začíná v obecnějších úrovních. V těch se zjistí, jestli vůbec existuje hledaná cesta a pokud existuje, přes které oblasti prochází. Poté se hledání přesune o úroveň níže a hledá se cesta skrze každý z *clusterů*, přes které by měla cesta procházet. Díky předpočítaným hraničním bodům je již známo, že taková cesta existuje. Tento postup se opakuje až do chvíle, kdy se hledání přesune na nejnižší úroveň a je nalezená kompletní cesta.

Velkou výhodou tohoto přístupu je i fakt, že výpočty lze rozložit na několik částí. Nejdříve stačí v obecnější rovině nalézt oblasti, kterými vede cesta. Poté již není potřeba prohledávat všechny konkrétnější úrovně naráz. Stačí, když víme, že cesta existuje, a že známe oblasti, kterými vede. Pro začátek je nalezena konkrétní cesta pouze přes *cluster* s počátečním bodem. Herní objekt se po ní může vydat, jelikož je jisté, že cesta vede správným směrem, a mezitím je možné dopočítat konkrétní podobu zbývající části cesty.

Tento přístup je výhodný zejména v *real-time* strategických hrách, kde je potřeba nalézt cestu pro větší množství herních objektů naráz natolik rychle, aby si uživatel nevšiml žádného zpomalení spojeného s potřebnými výpočty. Z toho důvodu je možnost rozložit výpočty do většího časového úseku vítaná.

4 Implementace vybraných algoritmů

Pro implementaci vybraných algoritmů byl použit programovací jazyk Java verze 7. Algoritmy s herním enginem komunikují pomocí rozhraní a nejsou na něm tedy příliš závislé. Bez větších problémů je lze upravit k použití v kterémkoliv jiném programu. To lze považovat za výhodu, v případě, že chceme demonstrovat funkčnost a zachovat jistou míru univerzality. Na druhou stranu, pokud by byla cílem maximální výkonnost a efektivita algoritmů, bylo by je potřeba optimalizovat přesně na míru daného programu, za důkladné znalosti všech jeho aspektů a na úkor použitelnosti v jiných aplikacích.

4.1 Generování terénu

Pro algoritmus generování terénu byla vytvořena samostatná třída nazvaná *MapGen*. Tato třída obstarává veškerou funkcionalitu týkající se generování mapy. Hlavní metodou této třídy je metoda *void generate(int Iter, MapGenI mapI, MapType type)*, která vygeneruje mapu terénu podle zadaných parametrů. Prvním parametrem této metody je počet iterací algoritmu, určující velikost vygenerované mapy. Dalšími parametry jsou rozhraní herní mapy a typ mapy, který má být generován.

Aby byla umožněna kompatibilita této třídy s herním enginem, je potřeba mít možnost vzájemné komunikace, konkrétně pak, aby si generátor mapy „rozuměl“ s herní mapou herního enginu. K tomuto účelu bylo vytvořeno rozhraní pro herní mapu, které obsahuje některé důležité metody. Rozhraní je nazváno *MapGenI* a obsahuje metody pro zjištění velikosti mapy a také pro nastavení typu terénu na určitou pozici na mapě. K popsání jednotlivých druhů terénu na mapě byla vytvořena enumerovací třída nazvaná *Terrain*.

Samotné vytváření mapy funguje velice jednoduše. Stačí vytvořit třídu implementující rozhraní *GameMapI* a tuto třídu předat metodě *generate(…)* spolu s typem mapy, který má být generován a počtem iterací algoritmu. Metoda *generate(…)* je hlavní metodou, která vygeneruje informace o mapě a vloží je do poskytnutého objektu implementujícího rozhraní *MapGenI*. Metoda samotná se skládá z několika logických celků, a každý z těchto celků obstarává jinou část algoritmu.

První částí je jeho inicializace. Zde se nastavují všechny proměnné objektu, vytváří se další pomocné objekty, jako je třeba objekt třídy *Random* obstarávající generování náhodných čísel a alokuje se a inicializuje pomocné pole hodnot, se kterým algoritmus počítá.

Toto pole má stejné rozměry jako budoucí mapa a je vytvořeno z primitivního datového typu *byte*. Aby bylo možné algoritmu vnutit předdefinované oblasti, je potřeba určit jednu hodnotu jako výchozí. Jakákoliv jiná hodnota je poté považována za předem předpřipravenou a algoritmus ji nebude přepisovat nově vypočítanou hodnotou.

Další částí algoritmu, která je do jisté míry pouze volitelná, je předdefinování typu mapy. K tomuto účelu slouží třída *MapTypeFactory*, které je předáno pole reprezentující mapu a třída *Params* obsahující parametry ovlivňující generování. Každá z metod třídy *MapTypeFactory* nastaví parametry generování a vloží předpřipravené hodnoty do pole, čímž určí charakteristiku vygenerovaného terénu.

V tuto chvíli přichází na řadu hlavní smyčka algoritmu, ve které probíhají všechny výpočty. Ta se skládá ze tří do sebe vnořených cyklů. První z nich realizuje iterativní verzi postupného vypočítávání nových „čtverců“ a postupně prochází všechny jejich velikosti.

```
Pro N od 0 do počtu iterací-1:
      Vypočítej počet čtverců v jedné řadě
      Vypočítej velikost kroku mezi čtverci
      Pro každý čtverec aktuální velikosti:
            Proveď square step
            Proveď diamond step
```

Počet čtverců v jedné řadě, což je hodnota, podle které se bude řídit další cyklus, vzrůstá po každé iteraci o dvojnásobek na hodnotu 2^x, kde x je počet iterací počínaje nulou. Místo použití funkce *pow()* z matematické knihovny by šlo inicializovat tuto proměnnou před začátkem cyklu na hodnotu jedna a na konci cyklu vždy vynásobit dvěmi. Toto řešení je sice méně výpočetně náročné, ale stejně tak je i méně čitelné. Z důvodu relativně nízkého počtu průchodů cyklem, u kterého se rozdíl nestačí projevit, byla upřednostněna čitelnost.

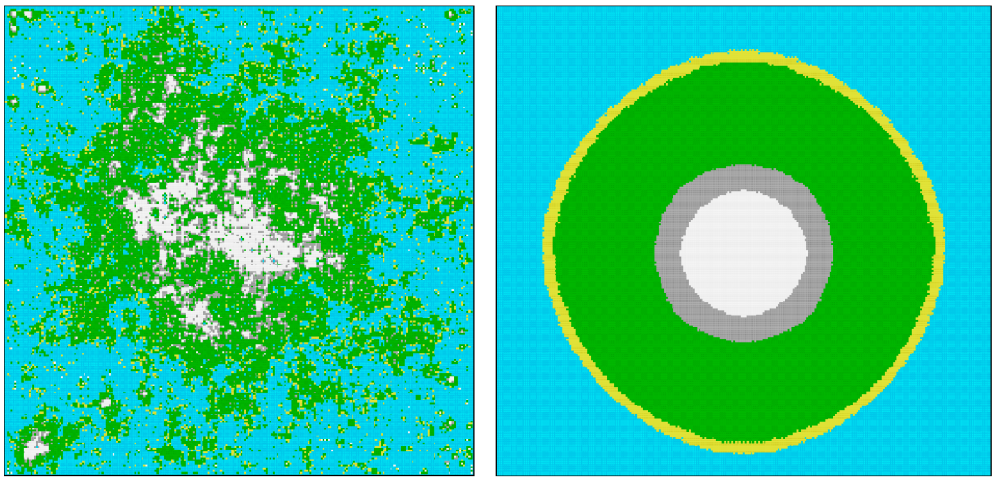
Poté, co je vypočten počet čtverců a velikost kroku mezi nimi, přichází na řadu samotné výpočty. O ty se starají dva do sebe vnořené cykly, které postupně prochází všechny čtverce a rozdělují je na nové.

Uvnitř těchto dvou cyklů jsou obsaženy výpočty po každý čtverec. Jako první se počítá takzvaný *square step*. Ten vypočítá hodnotu uprostřed aktuálního čtverce jako průměr ze všech jeho rohů a přidáním náhodné hodnoty.

Tato náhodná hodnota se pohybuje v určitý mezích a je potřeba ji dynamicky ovlivňovat pomocí regulační proměnné, která se může měnit v závislosti na různých okolnostech, většinou však podle velikosti aktuálního čtverce nebo množství provedených iterací. Toho je dosaženo tím, že je na začátku pevně daný rozsah náhodné hodnoty a při každé iteraci je vynásoben hodnotou danou vztahem:

$$2^{-h} \qquad (6)$$

Kde h je regulační proměnná určující hrubost vygenerovaného terénu a jedná se o kladné číslo. Tato hodnota je inkrementována každou iteraci o pevně danou hodnotu. Z rovnice (6) plyne, že zvyšováním hodnoty h klesá výsledná hodnota od jedné až k nule v nekonečnu a to pro kladná čísla včetně nuly. Změna hodnoty h tedy určuje, jak rychle má ztrácet náhodná hodnota vliv.



Obrázek 13: Vlevo je ukázka terénu vygenerovaného bez regulace hrubosti.
Vpravo je ukázka pravidelnosti terénu bez přidání náhodné hodnoty ke generovaným hodnotám.
Obě ukázky vznikly ze stejných počátečních hodnot určených pro tvar ostrova.

Dalším krokem je *diamond step*. Během tohoto kroku se dopočítávají hodnoty uprostřed každé strany aktuálního čtverce. Ty se získávají průměrem z rohů čtverce, který je pootočený o 45°. Dva z rohů pootočeného čtverce jsou zároveň rohy aktuálního čtverce na právě počítané hraně. Zbylé dva rohy jsou body vypočítáné během předchozího *square stepu* a jsou to středy čtverců sdílejících počítanou hranu.

Po této fázi nastává závěrečná část algoritmu, která má za úkol převzít vygenerovanou výškovou mapu a vytvořit z ní mapu terénu. Rozdělení výškové mapy lze provést jednoduchou analogií s reálným světem. Výšková mapa je rozdělena na výškové úrovně, kde každá z nich odpovídá jinému terénu. Výškové úrovně není potřeba dělit pouze podle explicitně dané hodnoty. Často je výhodnější zjistit tuto hodnotu dynamicky podle toho, kolik procent terénu má být nižší. Je-li nejnižší úrovní vodní hladina a má-li být mapa zpoloviny pod vodou, nastavíme její úroveň na 50%. Bude-li přidána další úroveň, musí být její úroveň vyšší než předchozí, jinak se nový terén vůbec neobjeví.

Tohoto dynamického zjišťování úrovně lze dosáhnout tím způsobem, že je vytvořen pomocný seznam, do kterého jsou uloženy všechny hodnoty vygenerovaného šumu. Když je tento seznam seřazen, je možné z něj zjistit, kterou hodnotu musíme zvolit, aby úroveň terénu odpovídala zadané procentuální hodnotě. Rozdělení terénu tak, aby z něj bylo 20% vody, se docílí tak, že zjistíme hodnotu ve 20% seznamu a použijeme ji jako výškovou úroveň. Všechny pozice s nižší hodnotou pak budou označeny za vodní hladinu. Obdobným způsobem si můžeme rozdělit mapu na libovolný počet různých výškových úrovní.

Nakonec je potřeba nastavit správné hodnoty terénu do samotného herního enginu. K tomu lze využít připraveného rozhraní *MapGenI* a jeho metody *void setTerrain(Terrain t, int x, int y)*. Té se předá typ terénu, jenž je zvolen podle toho, do které úrovně terénu patří a jeho pozice na mapě terénu.

4.2 Hledání cesty přes mapu

Hledání cesty přes mapu zajišťuje třída *AStar* nazvaná podle algoritmu, který je v ní implementován. Hlavní metoda této třídy je *List<Point> getWaypoints(GameObject go, Point dest)*, která dostane jako parametry herní objekt, pro který je potřeba najít cestu a cílový bod. Třída *Point* je třídou samotného herního enginu a obsahuje *x*-ovou a *y*-ovou souřadnici. Návratovou funkcí metody je pak seznam všech bodů nalezené cesty, seřazené v pořadí, ve kterém je herní objekt může postupně procházet směrem k cíli. Pro reprezentaci jednotlivých uzlů v mapě je vytvořena třída *Node*. Tato třída obsahuje hodnoty $G(n), H(n)$ i $F(n)$ a poskytuje metody pro jejich výpočty. Více informací v kapitole 3.3.

Instance třídy *AStar* vzniká již při inicializaci celé aplikace. Konstruktor třídy očekává instanci třídy implementující rozhraní herní mapy *GameMapI*, které umožňuje komunikaci s třídou implementující herní mapu v herním enginu. Navíc je možné konstruktoru ještě předat instanci grafického debugeru, který umožňuje vizualizovat různé informace o výpočtech algoritmu a jeho funkčnosti. Na Obrázku 14 jsou například zobrazeny všechny prohledávané uzly.

Obrázek 14: Ukázka grafického znázornění fungování algoritmu A*.
Prohledávaná pole jsou označena šedou barvou, pole nalezené cesty jsou vyznačena barvou červenou.

Rozhraní herní mapy deklaruje kromě zcela nepostradatelných funkcí, jako je zjištění terénu pomocí funkce *Terrain getTerrain(…)*, nebo zjištění schůdnosti funkcí *boolean isWalkable(…)*, i funkce spojené s tímto konkrétním enginem. Ten umožňuje například různé rozlišení mapy pro herní objekty a mapy pro terén. Rozhraní herní mapy tedy deklaruje i metody spojené s touto funkcionalitou, jako jsou metody *int getPathFHeight()* a *int getPathFWidth()*, které vrací výšku a šířku mapy herních objektů. Další metodou je *int GSqToNodeRatio()*, která vrací poměr mezi stranou mapy herních objektů a stranou mapy terénu.

Poté, co je zavolána metoda *getWaypoints(…)*, se spouští samotný algoritmus hledající cestu. Nejprve je potřeba vytvořit pomocné struktury a třídy, jenž algoritmus využívá. Dvěmi nejdůležitějšími jsou objekty obsahující množinu všech již prohledaný uzlů, tzv. *closed set*, a druhý, obsahující množinu všech uzlů, které jsou ještě určeny k rozgenerování, tzv. *open set*.

Množina *open* slouží k uchovávání všech uzlů určených k rozgenerovávání. Další uzel, který má být rozgenerován, se vybírá podle nejlepšího ohodnocení. Jako nejlépe ohodnocený uzel se bere uzel s nejmenší hodnotou $F(n)$. Z toho důvodu je jistě výhodné, aby požadovaná datová struktura měla co nejvýhodnější výběr prvku s nejlepším ohodnocením.

Pro implementaci množiny *open* byla využita vestavěná datová strukturu jazyka Java, prioritní fronta – *PriorityQueue*. Ta je implementována pomocí prioritní haldy – *priority heap*. Výhodou prioritní fronty je složitost výběru uzlu s nejlepším ohodnocením, které je $O(log(n))$ včetně vyjmutí uzlu ze struktury a jejím opětovném přeskupení. Nejlépe ohodnocený uzel je vždy na prvním místě a dostaneme se k němu se složitostí $O(1)$. Za logaritmickou složitostí stojí potřeba přerovnat všechny uzly poté, co je uzel s nejlepším ohodnocením odebrán. Stejnou složitost, $O(log(n))$, poskytuje i pro vložení nového uzlu. Zjištění přítomnosti uzlu je prováděno s lineární složitostí $O(n)$ [29].

K tomu, aby bylo možné použít prioritní frontu, je také potřeba určit, jakým způsobem se mají prvky řadit. K tomu lze využít rozhraní *Comparator<>*, které deklaruje metodu *compare(prvek1, prvek2)*, která porovná dva prvky a vrací kladné číslo v případě, že druhý prvek má větší hodnotu než první, nulu, pokud jsou prvky stejné a záporné číslo, pokud má větší hodnotu první prvek.

Toto rozhraní je implementováno ve třídě *NodeComparator*, která implementuje metodu *compare(Node n1, Node n2)*, která vrací jako lépe ohodnocený uzel takový uzel, který má menší hodnotu $F(n)$. Instance této třídy je předávána konstruktoru prioritní fronty, která se podle něj řadí. Uzel s nejmenší hodnotou $F(n)$ je pak vždy prvním prvkem.

U množiny *closed* oproti tomu potřebujeme co nejrychleji zjistit, zdali neobsahuje právě rozgenerovaný uzel. Efektivita této části algoritmu je kritická a při špatně zvolené datové struktuře může jít o citelné zpomalení celého algoritmu. To je znatelné s rostoucím objemem uzlů při hledání v nepříznivém terénu, kde velikost množiny *closed* rychle narůstá.

Množina *closed* je implementována pomocí vestavěné datové struktury jazyka Java, *HashSet*. Tato datová struktura zaručuje konstatní rychlost $O(1)$ pro metodu *contains()*[30], která zjišťuje přítomnost prvku v množině. Jiné datové struktury, které nejsou založeny na hashovací tabulce nebo přímém přístupu k prvku, mají rychlost operace *contains()* v nejlepším případě logaritmickou, a to u struktur založených na stromové struktuře. U datových struktur založených na struktuře seznamu to je dokonce lineární složitost [30].

Kromě vytvoření těchto objektů je také potřeba inicializace několika dalších. Mezi ně patří seznam *List<Point> waypoints*, do kterého se ukládají všechny body nalezené cesty, a který metoda vrací jako svou návratovou hodnotu. Dále je potřeba vytvořit ze zadaných souřadnic cílový a výchozí uzel. Navíc je vytvořen uzel *Node current*, který bude obsahovat právě rozgenerovávaný uzel. Před začátkem výpočtu je do něj přiřazen počáteční uzel, neboť se jedná o první uzel, který bude rozgenerován.

Po inicializaci přichází na řadu hlavní smyčka metody, obsahující výpočty. Ta opakuje svůj cyklus tak dlouho, dokud se právě rozgenerovávaný uzel nerovná cílovému. Pokud není cílovým uzlem, rozgeneruje všechny jeho sousední uzly a ty, na které je možné se pohnout a nejsou již v množině *closed*, přidá do množiny *open*. Nakonec je vybrán uzel s nejmenší hodnotou $F(n)$ k dalšímu rozgenerování a cyklus se opakuje.

```
Dokud není aktuální uzel shodný s cílovým:
      Rozgeneruj všechny sousední uzly
      Ty z nich, které nejsou v množině closed
      a zároveň je možné se na ně pohnout:
            Přidej do množiny open
      Vyber nový uzel s nejnižší hodnotou F k rozgenerování
```

Rozgenerovávání sousedních uzlů je prováděno pomocí dvou cyklů vnořených do sebe, s tím, že je nutné zohlednit poměr mapy herních objektů a mapy terénu. Krokem, o který cykly čítají, je pak právě tento poměr. To, že tento poměr bude celé kladné číslo, zajišťuje sama aplikace, kde je nastaven přímo poměr mezi těmito mapami.

Během rozgenerovávání je potřeba ohlídat okraje mapy, aby algoritmus nehledal do nekonečna. Stejně tak jsou vyřazeny uzly, na které není možné vstoupit. Také je výhodné vynechat uzel ze kterého se rozgenerovává okolí. Ten by byl vyřazen, jelikož ho již množina *closed* obsahuje a ušetří se tím výpočetní čas. Poté je je již vytvořena nová instance třídy *Node* pro právě rozgenerovaný uzel, který obsahuje své souřadnice a odkaz na předchozí uzel, z nějž byl rozgenerován.

U nově vytvořeného uzlu je potřeba zjistit, jestli ho již neobsahuje množina *closed*. Pokud tomu tak není, spočítají se hodnoty $G(n)$ a $F(n)$. K tomu slouží metoda *void updateVals(…)* třídy *Node*. Nejdříve je vypočtena hodnota $G(n)$, která je spočítána jako $G(n)$ hodnota předchozího uzlu, ke které je přičtena cena přechodu. Mapa však umožňuje pohyb všemi osmi směry, je tedy potřeba rozlišit cenu přechodů přímých a přechodů po diagonále. Pokud by cena přímých přechodů byla rovna jedné, cena přechodů po diagonále by byla dle pythagorovi věty odmocnina ze dvou. Počítat tuto hodnotu stále dokola však nemá smysl a tak je předpočítána dopředu. Dalším prvkem, který zpomaluje výpočty jsou aritmetické operace s čísly s desetinnou čárkou. Výhodnější je použití celých čísel. Aby bylo možné i nadále použít pro diagonální přechod odmocniny ze dvou, je zaokrouhlena, a všechny hodnoty přechodů jsou vynásobené konstantou. Odmocninu ze dvou je vhodné zaokrouhlit na tolik desetinných míst, aby po následném vynásobení konstantou byla desetinná část čísla nulová. Tuto konstantu je z důvodů lepší čitelnosti a pozdějších úprav kódu vhodné volit jako mocninu deseti.

Touto úpravou odpadne nutnost počítat s čísly s desetinnou čárkou. Je však potřeba tuto úpravu zohlednit ve všech částech algoritmu, jako třeba i při výpočtech heuristiky.

Poté, co je spočítána hodnota $G(n)$, je potřeba spočítat i hodnotu $H(n)$, a jejich sečtením získat konečnou hodnotu $F(n)$ daného uzlu. Pro výpočet heuristiky je použita *Manhattanská metrika*. Cesta nalezená s touto heuristikou sice vizuálně nekopíruje směr nejpřímější cesty a nevypadá tedy optimálně, ale tento problém lze řešit funkcí vyhlazující nalezenou cestu.

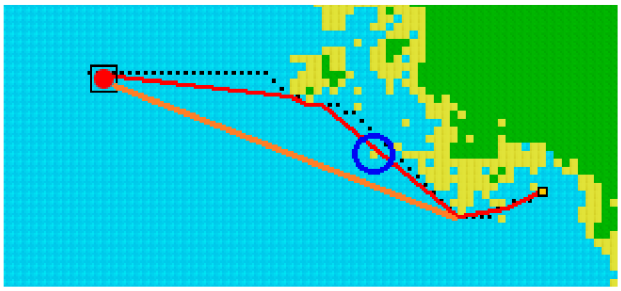
Uzel s již vypočítanými hodnotami se přidává do množiny *open*. Pokud již tento uzel množina *open* obsahuje, je potřeba porovnat, jestli nově vkládaný uzel nemá lepší ohodnocení hodnoty $G(n)$. To by znamenalo, že cesta nalezená přes předchůdce tohoto uzlu je výhodnější a je potřeba nahradit nově vkládaným uzlem ten starý. O to se stará funkce *void addUniqueBetter(…)*.

Když jsou všechny sousední uzly rozgenerovány, z množiny *open* se vybere nový uzel k rozgenerování, a to uzel s nejlepší hodnotou $F(n)$. Je-li množina *open* prázdná, znamená to, že neexistuje už žádné neprozkoumané místo, a že algoritmus nenalezl řešení. V této situaci je potřeba si vybrat, jak zareagovat. Jednou možností je nechat stát herní objekt na místě, s tím, že neexistuje žádná cesta k požadovanému cíli. Druhou možností, která je přirozenější, je nalézt nejlepší možnou alternativu, vedoucí do nejbližšího přístupného bodu. Tím je bod s nejmenší hodnotou heuristiky. Z důvodů funkčnosti herního enginu je též potřeba změnit souřadnice cílového bodu na souřadnice tohoto bodu.

4.2.1 Post-processing nalezené cesty

Ať už vyčerpáním uzlů z množiny *closed* a přepočítáním cesty k novému uzlu, nebo dosažením cílového uzlu, algoritmus již našel celou cestu. Ta je však pouze ve formě cílového uzlu a následných odkazů na předchozí uzly, které jsou navíc pouze na úrovni mapy terénu. K nalezení cesty na úrovni mapy herních objektů je zapotřebí za pomoci již nalezené cesty dopočítat její finální podobu. V této části algoritmu dochází jak k nalezení finální cesty, tak k jejímu vyhlazení a vyrovnání nerealistických tvarů způsobených funkčností algoritmu *A**.

Postup k nalezení a vyhlazení nové cesty na úrovni herních objektů je iterativní proces, který se vždy snaží najít další uzel, ke kterému lze dojít přímou cestou. Narazí-li na uzel, ke kterému již přímá cesta nevede, nahradí odpovídající část cesty posledním nalezeným přímým úsekem a začíná z jeho konce hledání nového úseku. Tento postup však není bezchybný.



Obrázek 15: Nalezená cesta (červeně) se bez dopředného prohledávání odklání od přímější cesty (oranžově). Modrý kroužek označuje bod způsobující tento odklon.

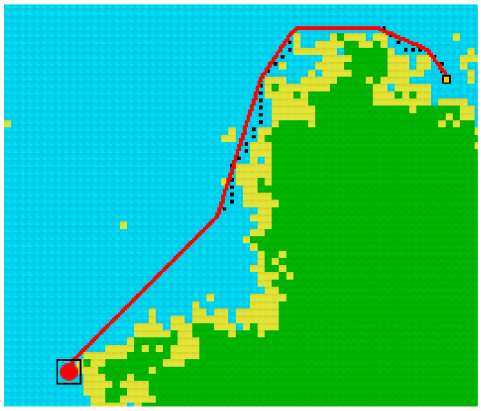
Jak lze vidět na Obrázku 15, při nevhodných podmínkách algoritmus nenalezne nejpřímější cestu, i když taková existuje. To je způsobeno tím, že v první chvíli, kdy se mu nepodaří spojit přímkou dva body, tak ukončí daný úsek a začíná s hledáním nového. Oblast, která blokuje přímou cestu mezi dvěma body (v modrém kolečku na Obrázku 15), lze však obejít i z druhé strany. Aby šlo nalézt i tyto úseky, ke kterým by se původní algritmus nedostal, je zavedeno takzavné *dopředné prohledávání*. To funguje na principu, kdy se při neúspěšně nalezené spojnici dvou bodů algoritmus

nepřesune na další úsek, ale vyzkouší část dalších uzlů, neexistují-li takové, ke kterým lze opět nalézt přímou cestu. Není-li nalezena žádná spojnice s některým z dalších uzlů, použije se poslední nalezený průchozí úsek.

Důležitou součástí algoritmu je funkce, která zjišťuje, jestli lze dva body spojit přímou cestou, která je celá průchodná. Ta je implementována jako funkce *List<Point> getLineWalkable(Point from, Point to, boolean onestep)*, která vrací buď seznam bodů spojujících krajní uzly, nebo hodnotu *null*, v případě, že taková cesta neexistuje. Parametry funkce jsou body, které mají být spojeny a boolean hodnota, která řeší některé krajní případy, které jsou popsány níže.

Zjištění bodů přímky mezi dvěmi body je implementováno upravenou verzí *Bresenhamova* algoritmu pro rasterizaci úseček. Ten je použit pro zjištění bodů aproximujících úsečku mezi krajními uzly. Algoritmus pracuje s body v mapě herních objektů, ale místo zobrazení vypočítaných bodů jsou tyto body předány funkci, která zjišťuje, jestli je tato pozice průchozí. Ve chvíli, kdy je nalezen neprůchozí bod, funkce vrátí hodnotu null. V opačném případě ukládá všechny body do seznamu, a při průchodnosti celé cesty vrací tento seznam s nalezenými body jako svou návratovou hodnotu.
Občas může nastat situace, kdy není nalezena cesta ani mezi dvěmi nejbližšími uzly. A to zejména v případech, kdy je umožněn diagonální průchod mezi dvěmi neprůchozími políčky terénu, které se dotýkají rohy. Při hledání přímého spojení těchto bodů může nastat situace, kdy jeden z bodů padne na neprůchozí oblast. V tu chvíli by se algoritmus nebyl schopný z této situace dostat. Jelikož víme, že tyto dva body jsou již nalezeny algoritmem A* v mapě terénu a tudíž mezi nimi existuje průchozí spojení, můžeme si dovolit aproximovat cestu mezi nimi přímkou i za tu cenu, že by některý z bodů mohl padnout na neprůchozí oblast. Zásah do této oblasti je však minimální a nijak neovlivní herní principy. Taková situace může vypadat třeba tak, že herní objekt přejde přes roh políčka terénu, místo aby ho o kousek dál obešel. Naopak snaha nalézt naprosto korektní cestu by mohla být neúměrně složitá, vzhledem k minimálním změnám cesty a prakticky nulovému vlivu na herní principy.

Poslední fází algoritmu je dokročení na závěrečný uzel cesty. Vzhledem k tomu, že cyklus se na konci posledního úseku přeruší, je na závěr potřeba ještě dokročit k tomuto bodu.



Obrázek 16: Nedokonalosti jsou možné i po zavedení dopředného prohledávání.
Jejich množství a zřetelnost je však nižší, než bez použití dopředného prohledávání.

V některých specifických případech stále nemusí vyhlazovací algoritmus najít nejrealističtěji vypadající cestu. Důvodem je situace, viditelná na Obrázku 16, kdy by pro nejpřímější spojení bylo nutné ukončit aktuální úsek ještě ve chvíli, kdy stále existují další uzly, které je možné spojit přímkou. Množství těchto nedokonalostí a jejich zřetelnost je však výrazně nižší než v situaci bez použití dopředného prohledávání. Ještě lepších výsledků je možné dosáhnout specializovanými algoritmy zaměřenými na prohledávání map, které umožňují pohyb jakýmkoliv směrem. Takovým algoritmem je například Theta*, o kterém pojednává práce [31].

5 Shrnutí výsledků

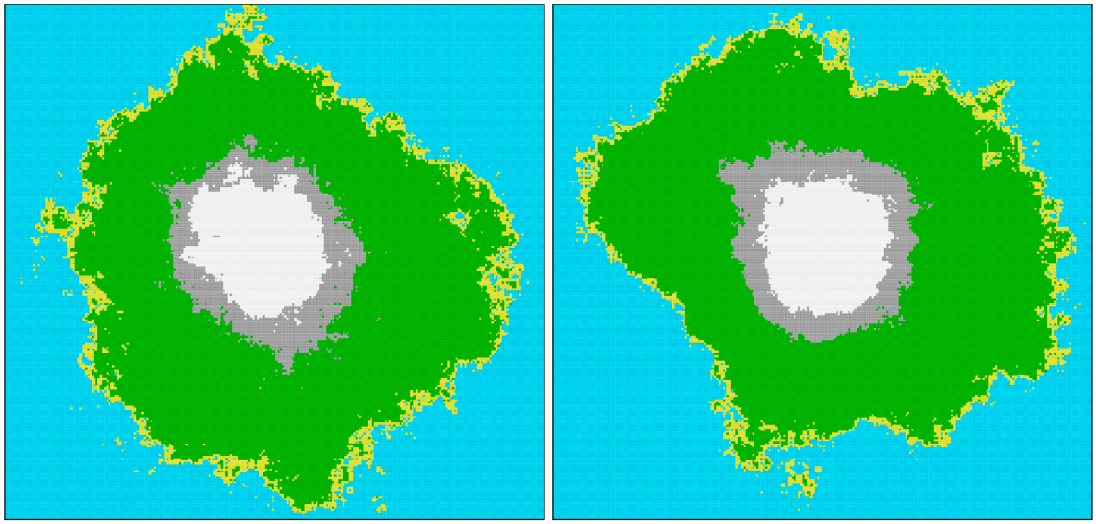
Za účelem otestování implementace algoritmů byla vytvořena aplikace, která vizualizuje dosažené výsledky. Tato aplikace využívá herní engine *BFST* tvořený autory Tomášem Černíkem a Štěpánem Karáskem. Tento herní engine poskytuje prostředí pro vykreslení mapy a herních objektů. Poskytuje také správu herních objektů a jejich pohybu.

Pro otestování dosažených výsledků byla zvolena ukázka několika typů map generovaných s různými parametry. Všechny vygenerované mapy se skládají z pěti druhů terénu: *vodní hladiny*, *pobřeží*, *travnatého terénu*, *skal* a *zasněženého terénu*. Do tohoto terénu jsou vygenerovány dva herní objekty. Jedním z nich je loď, která se pohybuje pouze po vodní hladině a druhým je pozemní jednotka, která se pohybuje po *travnatém terénu* a *pobřeží*.

5.1 Generování terénu

Prvním testovaným typem mapy je mapa, která má přednastavené hodnoty pro tvar ostrova, jak je ukázáno na Obrázku 17. Na tuto mapu jsou použity takové parametry, aby byl zachován tvar ostrova obklopeného vodou a zároveň pobřeží ostrova zůstalo členité. Maximální rozsah náhodné hodnoty je nastaven na ±50, tedy celkový rozsah je 100. Hodnota h regulující hrubost je inkrementována v každém cyklu o 0,45. Výška vodní hladiny je nastavená na 50 %, tudíž by měl ostrov zabírat zhruba polovinu mapy.

Největší změny mezi vygenerovanými mapami se projevují na tvaru ostrova, pobřežní linie a její členitosti, zatímco celková geometrie mapy zůstává stále velmi podobná.

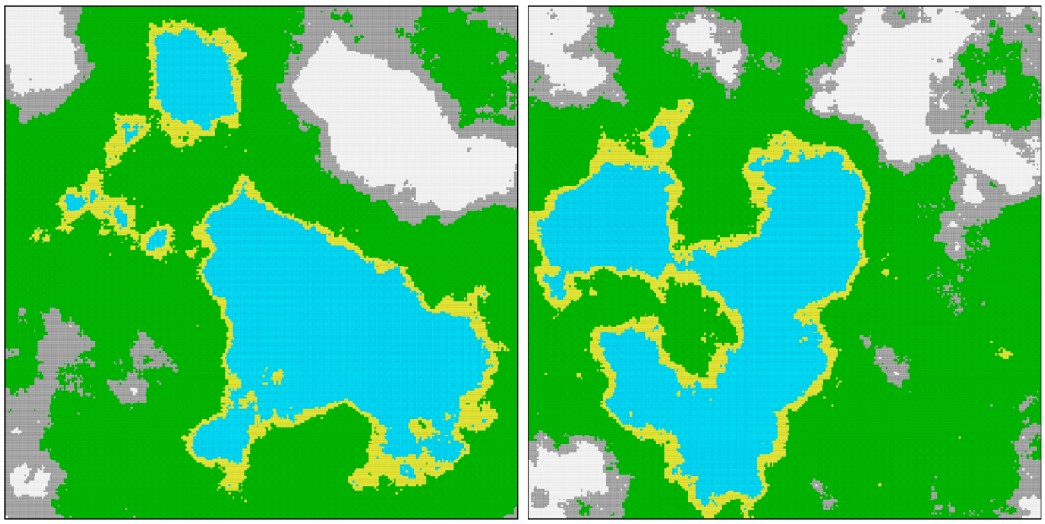


Obrázek 17: Mapa vygenerovaná ve tvaru ostrova obklopeného vodou.

Mapa v této podobě by mohla sloužit především v *real-time* strategických hrách, především pro hru dvou hráčů proti sobě a to především díky značné symetrii mapy. Každý z hráčů by mohl využít jednu stranu ostrova. Centrální neprůchozí oblast vytváří dvě oddělené cesty mezi hráči umožňující strategické manévrování. Zároveň lze využít i vodní plochu obklopující celý ostrov.

Druhou ukázkou je mapa s předdefinovanou vodní plochou v centrální části. Tento druh mapy opět používá předdefinované oblasti, ovšem liší se v hodnotách parametrů generování. Ty jsou nastaveny tak, aby mapa vykazovala větší variabilitu celkové geometrie a nižší detaily. Maximální rozsah náhodné hodnoty je navýšen na ±200. Díky tomu dochází k následnému efektu, kdy po přičtení náhodné hodnoty může dojít k přetečení, nebo podtečení hodnoty ve výškové mapě. To narušuje pravidelnost geometrie generované mapy a může vytvářet nečekané ostrovy, horská jezera nebo měnit celé oblasti mapy. Jelikož změny mají ovlivňovat geometrii mapy, ale úroveň detailů má být snížena, je potřeba velký rozsah náhodné proměnné regulovat přísněji, než u ostrovní mapy. Hodnota *h* regulující hrubost je inkrementována o 0,95. To je téměř dvojnásobná hodnota oproti ostrovní mapě. Také přechody mezi úrovněmi jsou méně členité.

Vodní hladina byla nastavena na spodních 20 % výškového rozsahu, aby nezabírala příliš moc místa na mapě. Naopak byly zvětšeny oblasti skal a sněhu, které pokryvají horních 20 % výškového rozsahu. Stejně tak byl částečně zvětšen rozsah pobřeží.

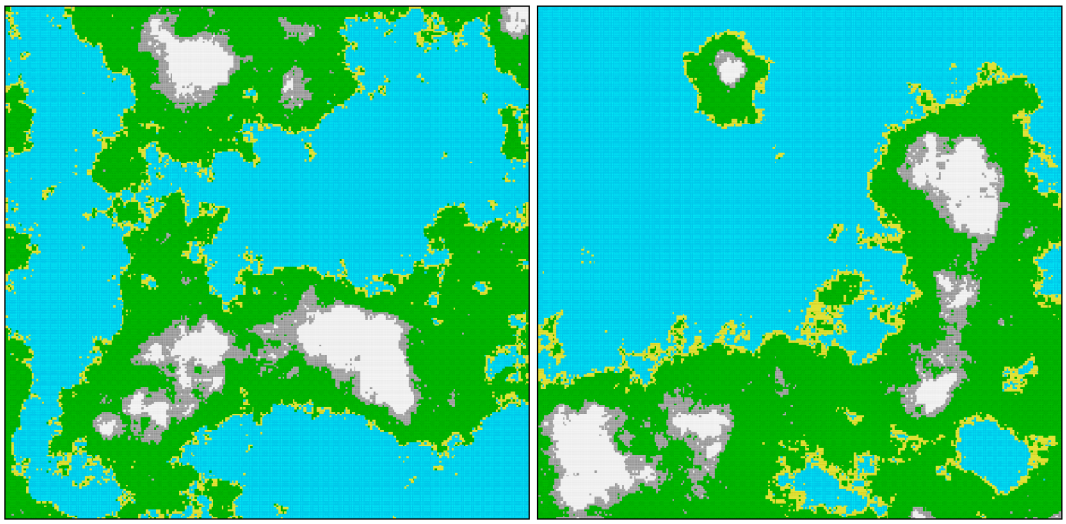


Obrázek 18: Přestože mapy vykazují podobné rysy, je zde vidět množství rozdílů.

Takto vytvořené mapy sice sdílejí podobné rysy, ovšem celková geometrie se může značně lišit, což je vidět i na obrázku 18. Z toho důvodu by takové mapy byly vhodné spíše pro *budovatelské* strategické hry pro jednoho hráče, ve který není potřeba zaručit více hráčům vyrovnané podmínky.

Třetím ukázkovým typem map je zcela náhodná mapa bez předdefinovaných oblastí. Aby byly tyto mapy co nejvíce náhodné a vykazovaly větší míru členitosti, je u tohoto typu mapy použit největší rozsah náhodné hodnoty, a to ±225. Také míra regulace je oproti předchozímu typu mapy snížena a hodnota *h* je inkrementována každou iteraci o 0,75. Vygenerovaná mapa vykazuje celkovou členitost způsobenou velkým rozsahem náhodné hodnoty i větší členitostí pobřeží, která je způsobena pomalejším inkrementováním regulační proměnné.

Vodní hladina byla nastavena na spodních 50 % výškového rozsahu. Jak si lze povšimnout na Obrázku 19, ve spojení s členitostí terénu působí mapa dojmem pohledu na velkou oblast z výšky. Tomu napomáhá i úzká pobřežní linie.



Obrázek 19: Tento typ mapy působí dojmem pohledu na větší oblast z výšky.

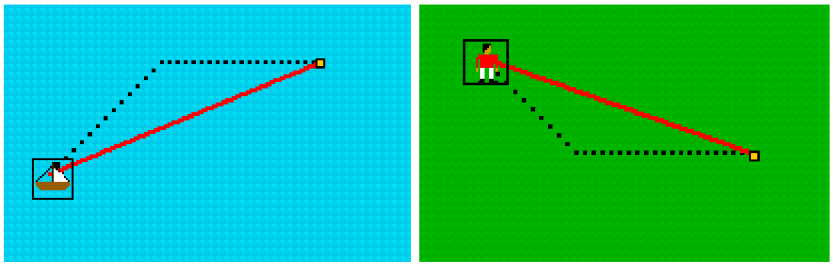
Zcela náhodně vytvořené mapy s těmito parametry jsou díky své členitosti vhodné především pro strategické hry simulující velké oblasti. Takovým typem strategických her může být například série *Civilization* [32].

5.2 Hledání cesty přes vygenerovanou mapu

Pro účely otestování algoritmu pro hledání cesty jsou do vygenerované mapy vloženy dva herní objekty. Jeden z objektů představuje loď, která se pohybuje pouze po vodní hladině. Druhým objektem je pozemní jednotka, která se pohybuje pouze po travnatém terénu a pobřeží.

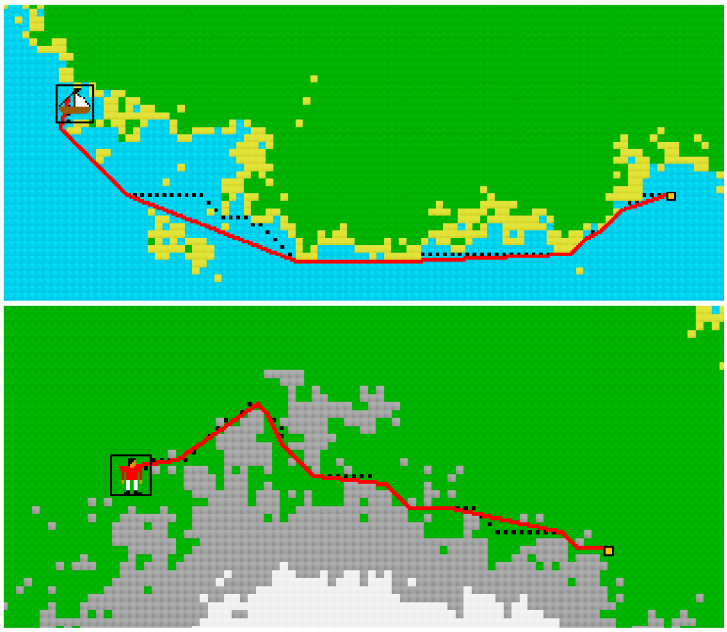
Aby šlo lépe testovat algoritmus hledání nejkratší cesty, bylo potřeba vizualizovat jeho výsledky. V ukázkové aplikaci lze zvolit tři možnosti vizualizace: vizualizace prohledávaného prostoru, vizualizace cesty nalezené v mapě terénu algoritmem A* a vizualizace finální cesty, po které se pohybuje herní objekt.

První situace, která může při hledání cesty nastat, je přímé spojení výchozího a cílového bodu bez překážek. V tomto případě nenastávají žadné problémové situace. Cesta je nalezena algoritmem A* a následně je vyhlazena. Jelikož v cestě neleží žádné překážky, vyhlazená cesta spojuje přímo koncové body, jak je vidět z Obrázku 20.



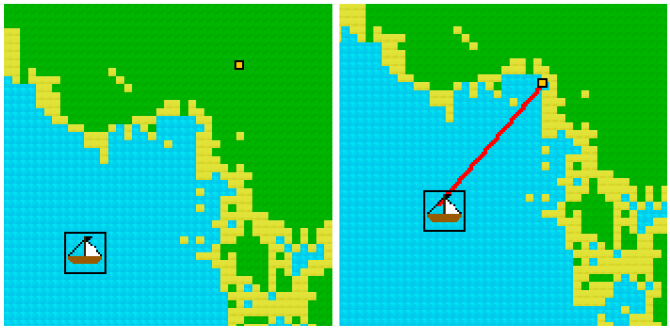
Obrázek 20: Algoritmus A* (černě) nalezne nejkratší cestu v mapě terénu.
Ta je vyhlazena a jsou dopočítány všechny body do mapy herních objektů (červeně).

Druhou situací, která může při hledání cesty nastat, je cesta vedoucí okolo různých překážek, kde neexistuje přímé spojení výchozího a cílového bodu, jak je ilustrováno na Obrázku 21. Cílový bod je však přístupný a existuje k němu dosažitelná cesta. Algoritmus A* nalezne cestu bez problémů. Při následném vyhlazování cesty může ojediněle docházet k situacím, kdy na některých úsecích není nalezeno nejpříměji vypadající spojení. Více informací lze nalézt v kapitole 4.2.1.



Obrázek 21: Cesty nalezené přes členitý terén algoritmem A* (černě) a jejich následné vyhlazení (červeně).

Třetí situací, která může při hledání cesty nastat, je nedostupný cílový bod. Algoritmus si musí umět poradit i s touto situací. Z pohledu strategické hry není vhodným řešením, aby herní objekt nereagoval, a proto musí být nalezen nový cílový bod, viz Obrázek 22. Ten je vybrán na základě nejmenší hodnoty heuristiky. Je to tedy bod s nejkratší odhadovanou vzdáleností od původního nedostupného bodu.



Obrázek 22: Pokud leží cílový bod na nedostupném místě, vybere se místo něj nový cílový bod.

Algoritmus A* byl schopný nalézat cesty vygenerovaným terénem. Jím nalezené cesty však z důvodu pohybu pouze v osmi směrech vizuálně nekopírovaly přímou cestu. Proto byl přidán algoritmus vyhlazování cesty, který zároveň umožnil získání cesty v mapě herních objektů z bodů nalezených v mapě terénu algoritmem A*. Spojení těchto algoritmů umožnilo zároveň nalezení i zrealističtění výsledné cesty.

6 Závěr

Procedurální generování mapy je z hlediska použitelnosti silně závislé na oblasti, pro kterou je použito. Zvláště v rámci strategických her je potřeba tuto možnost dobře promyslet a naplánovat s ohledem na všechny možné funkční prvky dané hry. Proces procedurálního generování mapy se skládá z teoreticky neomezeného množství algoritmů, z nichž má každý na starost jiný podsystém mapy. Výzvou pro vývojáře je správná kombinace odlišných algoritmů, z nichž každý má často specifický výstup.

Pro procedurální generování terénu byl vytvořen balík s názvem *bfst.map.mapGen*, obsahující soubory potřebné k integraci do herního enginu i pro samotné vygenerování mapy. K implementaci byl použit *diamond-square* algoritmus popsaný v teoretické části práce. Tento algoritmus byl úspěšně začleněn do herního enginu a použit ke generování map. Při použití algoritmu se projevila jeho schopnost výrazně regulovat výsledný terén a relativně lehce dosáhnout celé škály různých map vhodných pro různé účely. To lze demonstrovat na ukázkové aplikaci, kde je možnost vygenerovat odlišné druhy map.

Zároveň byl vytvořen balík s názvem *bfst.pathFinding*, obsahující soubory potřebné pro vyhledávání cesty v mapě. K implementaci byl použit algoritmus A*, který byl také popsán v teoretické části práce. Řešení, které je implementované v této práci, zvládá vyhledávání cesty ve vygenerované mapě a následné vyhlazení nerealistických tvarů produkovaných algoritmem A*. Použitím douúrovňové architekruty je také výrazně snížen počet prohledávaných uzlů. Algoritmy využívají ke komunikaci rozhraní a jsou tedy nezávislé na konkrétním herním enginu. Z toho důvodu je možné je použít v jakékoliv jiné aplikaci implementující jejich rozhraní.

Jako možné rozšíření algoritmů se nabízí u generování mapy přidání dalších prvků, a to jak krajinných, tak i prvků strategického rázu pracujícími s herními principy. Ty by musely být vázány na specifičtější určení funkčnosti dané aplikace.

Pro strategické hry v reálném čase by bylo zapotřebí dalších optimalizací algoritmu pro hledání cest, aby byl schopen reagovat na větší množství požadavků naráz bez znatelného zpoždění. Možnou cestou pro zvýšení výkonosti jsou hierarchické vyhledávací algoritmy či pre-processing vygenerované mapy a vytvoření dalších struktur s pomocnými informacemi.

Literatura

[1] TURNER, Martin. The origins of fractals. *Plus magazine: living mathematics* [online]. 1998 [cit. 2014-05-01]. Dostupné z: http://plus.maths.org/content/origins-fractals

[2] Fractals. *Pomegranate apps* [online]. [cit. 2014-05-01]. Dostupné z: http://www.pomegranateapps.com/fractals/

[3] WEISSTEIN, Eric W. Koch Snowflake. *MathWorld: A Wolfram Web Resource* [online]. [cit. 2014-05-01]. Dostupné z: http://mathworld.wolfram.com/KochSnowflake.html

[4] Fractals in nature. *Fractal foundation* [online]. [cit. 2014-05-03]. Dostupné z: http://fractalfoundation.org/category/natural-fractals/

[5] MILLIGAN-CROFT, David. Fractals in Nature. *Thereisnocavalry* [online]. 2012, 2014-08-09 [cit. 2014-05-01]. Dostupné z: http://thereisnocavalry.wordpress.com/2012/08/09/fractals-in-nature/

[6] Star Trek II: Khanův hněv. *CSFD: Česko-Slovenská filmová databáze* [online]. [cit. 2014-05-01]. Dostupné z: http://www.csfd.cz/film/20587-star-trek-ii-khanuv-hnev/

[7] Film Milestones in Visual and Special Effects. *Filmsite* [online]. [cit. 2014-05-01]. Dostupné z: http://www.filmsite.org/visualeffects11.html

[8] WELLONS, Christopher. Noise Fractals and Clouds. *Null program* [online]. 2007 [cit. 2014-05-01]. Dostupné z: http://nullprogram.com/blog/2007/11/20/

[9] Ken Perlin: vita. *New York university* [online]. [cit. 2014-05-01]. Dostupné z: http://mrl.nyu.edu/~perlin/doc/vita.html

[10] WEISSTEIN, Eric W. Simplex. *MathWorld: A Wolfram Web Resource* [online]. [cit. 2014-05-01]. Dostupné z: http://mathworld.wolfram.com/Simplex.html

[11] GUSTAVSON, Stefan. Simplex noise demystified. In: *Linköping University: Department of Science and Technology* [online]. 2005 [cit. 2014-05-03]. Dostupné z: http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

[12] ONG, Teong Joo, Ryan SAUNDERS, John KEYSER a John J. LEGGETT. Terrain Generation Using Genetic Algorithms. [online]. [cit. 2014-05-01]. Dostupné z: http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p1463.pdf

[13] FOURNIER, Alain, Don FUSSELL a Loren CARPENTER. Computer rendering of stochastic models. *Communications of the ACM* [online]. 1982, vol. 25, issue 6, s. 371-384 [cit. 2014-05-01]. DOI: 10.1145/358523.358553. Dostupné z: http://portal.acm.org/citation.cfm?doid=358523.358553

[14] MILLER, Gavin S P. The definition and rendering of terrain maps. *ACM SIGGRAPH Computer Graphics* [online]. 1986, vol. 20, issue 4, s. 39-48 [cit. 2014-05-01]. DOI: 10.1145/15886.15890. Dostupné z: http://portal.acm.org/citation.cfm?doid=15886.15890

[15] BEARD, Daniel. Terrain Generation: Diamond Square Algorithm. *Daniel Beard: iOs and other stuff* [online]. 2010 [cit. 2014-05-01]. Dostupné z: http://www.danielbeard.io/blog/2010/08/06/terrain-generation-and-smoothing

[16] AMIT, Patel. Polygonal map generation for games. *Red Blob Games* [online]. 2010 [cit. 2014-05-05]. Dostupné z: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation

[17] TEOH, Soon Tea. River and Coastal Action in Automatic Terrain Generation. In: *Proc. of the International Conference on Computer Graphics and Virtual Reality* [online]. 2008 [cit. 2014-05-01]. Dostupné z: http://www.cs.sjsu.edu/~teoh/research/papers/CGVR08_terrain.pdf

[18] WEST, Mick. Random Scattering: Creating Realistic Landscapes. *Intel Developer Zone* [online]. 2008 [cit. 2014-05-01]. Dostupné z: https://software.intel.com/en-us/articles/random-scattering-creating-realistic-landscapes

[19] CEPERO, Miguel. The forest. *Procedural World* [online]. 2011 [cit. 2014-05-01]. Dostupné z: http://procworld.blogspot.cz/2011/05/forest.html

[20] Risk (game). *Wikipedia* [online]. 2014-04-26 [cit. 2014-05-01]. Dostupné z: http://en.wikipedia.org/wiki/Risk_%28game%29

[21] Risk. *Wikimedia commons* [online]. 2013-03-09 [cit. 2014-05-01]. Dostupné z: http://commons.wikimedia.org/wiki/Risk

[22] ZBOŘIL, František a František ZBOŘIL. *Základy umělé inteligence* [elektronická skripta]. 2013, s. 16 a 33 [cit. 2014-05-01]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IZU/private/oporaizu-esf-5a.pdf

[23] WEISSTEIN, Eric W., Andreas LAUSCHKE a Len GOODMAN. Dijkstra's Algorithm. *MathWorld: A Wolfram Web Resource* [online]. [cit. 2014-05-02]. Dostupné z: http://mathworld.wolfram.com/DijkstrasAlgorithm.html

[24] SNEYERS, J., SCHRIJVERS, T. a DEMOEN, B. Dijkstra's algorithm with Fibonacci heaps: An executable description. In *In CHR. In 20th Workshop on Logic Programming (WLP'06)*. 2006. S. 182–191 Kap 2. The single-source shortest path problem.

[25] AMIT, Patel. Introduction to A*: From Amit's Thoughts on Pathfinding. *Stanford theory group* [online]. 2014-04-27 [cit. 2014-05-02]. Dostupné z: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[26] AMIT, Patel. Heuristics: From Amit's Thoughts on Pathfinding. *Stanford theory group* [online]. 2014-04-27 [cit. 2014-05-02]. Dostupné z: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

[27] PINTER, Marco. Toward More Realistic Pathfinding: Smoothing the A* Path. *Gamasutra* [online]. 2001 [cit. 2014-05-08]. Dostupné z: http://www.gamasutra.com/view/feature/131505/toward_more_realistic_pathfinding.php?page=2

[28] BOTEA, Adi, Martin M a Jonathan SCHAEFFER. Near optimal hierarchical path-finding. In: *Journal of Game Development*. 2004, s. 7-28. 1. DOI: 10.1.1.112.314.

[29] Priority Queue. *Oracle* [online]. [cit. 2014-05-02]. Dostupné z: http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

[30] Java Collections – Performance (Time Complexity). *Information Technology Gems* [online]. 2011 [cit. 2014-05-02]. Dostupné z: http://infotechgems.blogspot.cz/2011/11/java-collections-performance-time.html

[31] DANIEL, Kenny, Alex NASH, Sven KOENIG a Ariel FELNER. Theta*: Any-Angle Path Planning on Grids. In: *Journal of artificial intelligence research*. Menlo Park: Assn For Adv Artificial I, 2010, s. 533-579. ISBN 9781577355359. Dostupné z: http://jair.org/media/2994/live-2994-5259-jair.pdf

[32] *Civilization V* [online]. 2010 [cit. 2014-05-06]. Dostupné z: http://www.civilization5.com/

PŘÍLOHA A
Obsah CD

Přiložené CD obsahuje zdrojové kódy a soubory aplikace, spustitelnou verzi aplikace ve formátu .jar, bakalářskou práci ve formátu .pdf i se zdrojovým kódem a soubory s návodem k instalaci a k obsluze aplikace.

./src/ – zdrojové soubory celé aplikace, včetně herního enginu *bfst*
./build.xml – buildfile aplikace pro nástroj ant
./src/bfst/map/mapGen/ – zdrojové soubory vytvořené v rámci bakalářské práce, které jsou určené ke generování mapy
./src/bfst/pathFinding/ – zdrojové soubory vytvořené v rámci bakalářské práce, které jsou určené ke hledání cesty
./src/bfst/debug/ – zdrojové soubory vytvořené v rámci bakalářské práce, které jsou určené ke grafickému znázornění fungování algoritmu hledání cesty
./dist/bp.jar – spustitelná aplikace ve formátu .jar
./doc/ – složka obsahující bakalářskou práci ve formátu .pdf i její zdrojový kód
./readme.pdf – návod k ovládání aplikace
./install.txt – návod k přeložení zdrojových kódů