**Univerzita Hradec Králové**
**Fakulta informatiky a managementu**
**Katedra informatiky a kvantitativních metod**

# Procedurální generování v počítačové grafice

Bakalářská práce

Autor: Jan Flégl
Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Jakub Beneš

Hradec Králové Duben 2024

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 23.4.2024 Jan Flégl

# Abstrakt

## Název: Procedurální generování v počítačové grafice

Práce se zabývá představením Procedurálního generování v počítačové grafice. Problematika a téma procedurálního generování je v první části práce představena a vysvětlena.

V další části práce jsou již jednotlivé algoritmy představeny, a jejich fungovaní je vysvětleno. Vybranými algoritmy jsou Perlinův šum, Diamond-Square algoritmus, Wave Function Collapse a Voroného diagramy. Všechny zmíněné algoritmy hrají svou klíčovou roly v procesu generování terénu a v praxi jsou hojně využívány.

Na základě implementace jsou vybrané algoritmy mezi sebou porovnány a poté je zhotoveno závěrečné vyhodnocení jednotlivých algoritmů a jejich porovnání.



**Klíčová slova:** Perlin noise, procedurální generování, OpenGL, Diamond-Square algorithm, Wave Function Collapse, Voronoi diagram, proceduraly generated terrain

# Abstact

## Title: Procedural generation in computer graphics

The thesis deals with the introduction of Procedural Generation in computer graphics. The issue and topic of procedural generation are introduced and explained in the first part of the work. In the next part of the work, individual algorithms are introduced, and their functioning is explained. The selected algorithms include Perlin Noise, the Diamond-Square algorithm, Wave Function Collapse, and Voronoi diagrams. All these algorithms play a key role in the terrain generation process and are widely used in practice.

Based on the implementation, the selected algorithms are compared with each other, followed by a final evaluation and comparison of the individual algorithms.

**Key words:** Perlin noise, procedural generation, OpenGL, Diamond-Square algorithm, Wave Function Collapse, Voronoi diagram, proceduraly generated terrain

# Obsah

# Seznam obrázků

# Seznam ukázek kódů



# Seznam matematických vzorců

# 1 Úvod

V rozsáhlém světě počítačové grafiky napříč odvětvími, od počítačových her až po medicínské aplikace či animované filmy, procedurální generování vstupuje do herního pole jako velice mocný nástroj umožňující vnímat a vytvářet prostředí jinak než doposud. Pomocí této techniky jsou vývojáři schopni vyobrazovat svět nekonečných možností, světy velice náhodné, ale zároveň velice podobné tomu našemu, reálnému.

Velká většina lidí narozena od přelomu 21. století až do současnosti hráli nějakou počítačovou hru. Jenom málo kdo by avšak řekl, že většina moderních her, ale samozřejmě i těch starších, jsou vytvářeny pomocí procedurálního generování. Vyjímaje již dobře známých a očividných her, jako je Minecraft, Terraria a česká hra Factorio, lze jmenovitě zmínit třeba velice populární hru Witcher 3, která používá techniku procedurálního generování pro generaci nejen terénu, ale i stromů a trávy. Bez předchozích zkušeností s hraním her, avšak s pevným základem v medicíně, může být zajímavé, že procedurální generování se používá při tvorbě anatomických modelů, tedy modelů orgánů, tkání a dalších anatomických struktur. Tyto modely jsou poté využívány pro výcvik nebo chirurgické plánování. Pro potřeby výcviku se poté vytvářejí i simulace operací a lékařských procedur.

Procedurální generování našlo využití v armádě a letectví, kde slouží především simulačním účelům boje, či v letectví za účelem výcviku, aby každý scénář byl odlišný a poskytl unikátní a neopakovatelnou situaci, se kterou se vojáci, či piloti mohou ve své profesi setkat.

Cílem bakalářské práce je prozkoumat metody procedurálního generování ve 3D scéně. Na základě informací a obtížnosti budou vybrány 2 známé algoritmy, které budou implementovány a porovnány dle vybraných kritérií.

První část bakalářské práce se bude zaměřovat na porozumění použitého grafického rozhraní OpenGL, ale také na možnosti použití jiných grafických API. Součástí této kapitoly je také vysvětlení základních geometrických primitiv a zobrazovacího řetězce.

Po této kapitole bude následovat rozbor, proč je výhodné používat procedurální generování a obecné představení problémů a výzev, které při tvorbě

takovýchto terénů nastávají. Následně budou prozkoumány známé algoritmy, které se používají pro tvorbu procedurálně generovaného terénu a poté této kapitole budou na vybrané algoritmy provedeny měření, jako jsou snímky za vteřinu či rychlost generování terénu. Tyto algoritmy budou mezi sebou porovnány a bude vyhodnoceno, který z nich je nejvhodnější.

Po měřící kapitole bude následně porovnány již měřené algoritmy, tak i neměřené a budou mezi sebou opět porovnány, avšak již ne podle číselných kritérií, ale podle obecných znalostí programování a počítačové grafiky, spolu s obecnými znalostmi ze světa IT.

# 2 OpenGL

Open Graphic Library, ve zkratce OpenGL, představuje základní stavební kámen v oblasti počítačové grafiky. OpenGL, jako silné, multiplatformní a v oblasti počítačové grafiky známé grafické API, tedy application programming interface, česky aplikační programové rozhraní, umožňuje vývojářům vytvářet nejrůznější složité 3D, ale také 2D, grafické aplikace, které lze programovat v mnoho programovacích jazycích. Celá praktická část je vytvořena pomocí tohoto rozhraní, avšak ničemu nebrání využít i jiné, potencionálně náročnější, ale za to výkonnější, grafické rozhraní.

## 2.1 Historie OpenGL

„K pochopení, jak a proč bylo OpenGL vytvořeno, je důležité vzít na uváženou grafické rozhraní před samotným OpenGL“[1]. Grafická API hrají klíčovou roli ve vývoji počítačové grafiky, díky kterým umožňují vývojářům interagovat s grafickým hardwarem. Před samotným vznikem OpenGL v roce 1992, existovala řada dalších grafických rozhraní, která již v tu dobu formovala základy dnešní moderní počítačové grafiky. Graphical Kernel System, ve zkratce GKS, byl jedím z prvních významných grafických rozhraní vznikl již v roce 1985, které bylo určeno pro 2D grafiku a je známé především pro svoji univerzálnost a přenositelnost mezi různými systémy.

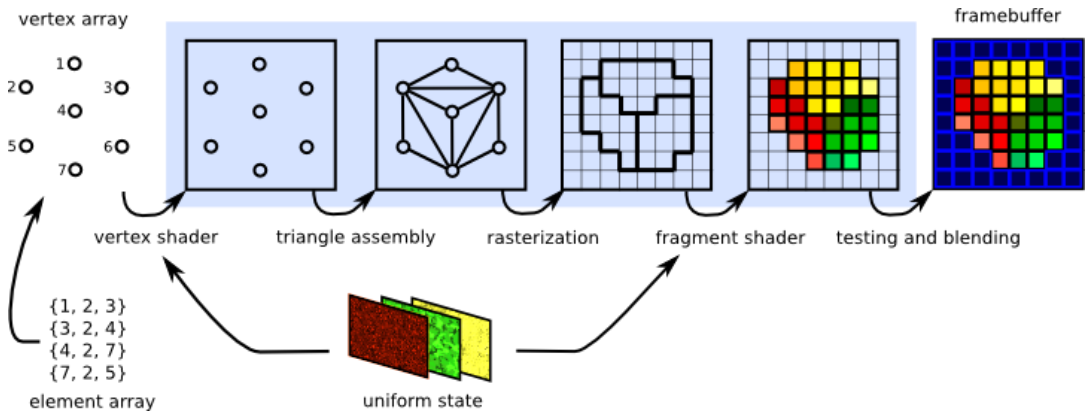
Dalším důležitým a významným krokem ve vývoji grafických API byl vznik PHIGS, tedy Programmer´s Hierarchical Interactive Graphics System, které bylo zavedeno do funkčnosti v osmdesátých letech minulého století a někdy se uvádí jako předchůdce samotného OpenGL. To představovalo mnohem pokročilejší možnosti než GKS, ale hlavně umožnilo vývojářům vytvářet komplexní 3D scény a objekty. Nehledě na to, jak přelomová tato technologie byla a ve své době byla považována za velice pokročilou, bylo postupem času překonáno z důvodu omezení ve výkonu.

## 2.2 Zobrazovací řetězec

Zobrazovací řetězec, známý také jako grafická pipeline, je klíčovým konceptem pro pochopení, jak jsou grafická data zpracovávána a transformována na obraz, který je zobrazen na monitoru.

Na začátku do řetězce jsou dodána geometrická data. „Ta mohou být ve formě polygonální sítě nebo jako různé druhy plátů či povrchy vyšších řádů“ [2]. Na tyto jednotlivé body je poté aplikován vertex shader, jenž je vysvětlen v kapitole níže. Před jeho aplikací je ovšem nezbytné, aby geometrická data byla správně připravena a optimalizována pro zpracování. K tomuto účelu se používají Vertex a Index buffery.

Po aplikace vertex shaderu jsou jednotlivé, již nově transformované vrcholy převedeny do sítě trojúhelníků a tyto trojúhelníky jsou poté rasterizovány. „Rastrizace zahrnuje určení pixelů, které daný primitiv pokrývá na obrazovce, aby bylo možné vypočítat barvy každého pokrytého pixelu.“[3]. Po rasterizaci je poté využita funkce fragment shaderu. Po fragment shaderu je téměř již hotovo, zbývá pouze udělat tzv. „depthTest“, tedy hloubkový test pomocí Z-Bufferu. Ten porovnává hodnotu hloubky jednotlivých pixelů a určuje, který má být v popředí a který v pozadí. Výsledkem Z-Bufferu tedy bude realisticky vypadající scéna s hloubkou, která je již vykreslena uživateli na monitor.




**Obrázek 1 - Zobrazovací řetězec [3]**

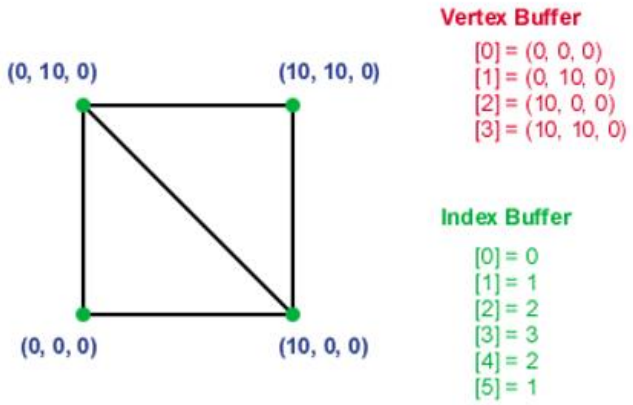
### 2.2.1 Index buffer

Tento buffer obsahuje indexy, které odkazují na pozice vrcholů uložených ve Vertex bufferu. Jednou z dominantních funkcí index bufferu je optimalizace

ukládání a zpracovávání geometrických dat tím, že umožňuje opakované použití vrcholů při definování grafických primitiv, jako jsou třeba trojúhelníky, bez nutnosti jejich duplikace. Tento proces tedy snižuje množství dat potřebných k uchování a zpracování, a tedy to vede k rychlejšímu zpracování scény.

### 2.2.2 Vertex buffer

Vertex buffer je základní datová struktura používaná v počítačové grafice pro ukládání geometrických informací o vrcholech, které tvoří 3D objekty. Tento buffer obsahuje seznam vrcholů, kde každý vrchol může zahrnovat různé typy dat, jako jsou pozice, barvy, souřadnice textury a normály.




**Obrázek 2 - Vertex a index buffer [4]**

## 2.3 Shadery

V moderní době počítačové grafiky se již dávno upustilo od tradiční fixní pipeline a přešlo se k mnohem výkonnější a flexibilnější metodě, shaderům, která poskytuje programátorům mít plnou kontrolu nad grafickým výstupem. Na rozdíl od fixní pipeline, shadery umožňují vývojářům a grafikům pracovat na té nejnižší možné úrovni, jelikož tyto skripty běží přímo na GPU, grafických procesorech, a nabízejí detailní zpracování vertexů, pixelů a dalších grafických primitiv. Pomocí shaderů tedy vzniká možnost vytvářet pokročilé vizuální efekty a interaktivní grafické aplikace v reálném čase. Shadery mají i speciální programovací jazyk, pro OpenGL a Vulcan je to GLSL, OpenGL Shading Language, a pro DirectX je to HLSL, tedy High-Level Shading Language.

### 2.3.1 Vertex shadery

Vertex shadery jsou klíčovým komponentem v moderní 3D grafické pipeline, sloužící k manipulaci s vrcholovými daty. Každý vertex shader je aplikován na jednotlivé vrcholy scény a jeho hlavním úkolem je transformovat pozice vrcholů z modelových souřadnic do souřadnic clip prostoru. Tento proces obnáší řadu transformací, včetně modelové transformace, která přemisťuje objekty v prostoru, pohledové transformace, jež definuje umístění a orientaci kamery, a projekční transformace, která určuje, jak jsou objekty promítnuty na obrazovku. Výstupem každého vertex shaderu je tedy nová pozice jednotlivých vrcholů.

Vertex shadery však nejsou omezeny pouze na transformaci pozic vrcholů. Mají schopnost modifikovat nebo generovat dodatečné per-vertex atributy, jako jsou barvy, normály a souřadnice textury. Tyto rozšířené atributy jsou následně předávány do dalších etap zpracování grafické pipeline, typicky k fragment shaderům. Fragment shadery tyto informace využívají k určení vlastností pixelů, které se nakonec zobrazí na obrazovce uživatele, umožňující tak složitější efekty, jako jsou osvětlení, stínování a aplikace textur. Vertex shadery tedy představují první krok v pokročilém zpracování grafických dat, umožňující vývojářům vytvářet detailní a vizuálně působivé 3D scény.

### 2.3.2 Fragment shadery

Fragment shadery, taktéž známé pod názvem pixel shadery, operují na tzv. fragmentech, což jsou jednotky dat vznikajících procesem rasterizace. „Víme, že operace rasterizace interpoluje veličiny, jako jsou barvy, hloubky a souřadnice textur. Fragment shadery využívají tyto interpolované hodnoty, stejně jako mnoho dalších druhů informací, k určení barvy každého pixelu fragmentu.“[5]. Přes všechny proměnné je tedy interpolováno pomocí rasterizace a následně jsou poslány do fragment shaderu, kde jsou s nimi prováděny jakékoliv operace, které fragment shaderu zadáme.

Všechny výpočty prováděné fragment shaderem jsou prováděny paralelně a na několika fragmentech, jejichž počet je ovlivněn grafickou kartou, která je k operacím používána.

Fragment shadery mohou být využity k vytvoření řady vizuálních efektů, například k simulaci realistických povrchů vody, skla nebo různých materiálů. V kontextu realistického renderování mohou například simuluovat způsoby, jakými světlo interaguje s různými povrchy, což vede k vytvoření realistických odrazů, refrakcí nebo stínů. V praxi to může zahrnovat komplexní algoritmy, které napodobují fyzikální vlastnosti světla a materiálů.

### 2.3.3 Tessellation shadery

Tessellation shadery umožňují grafické jednotce dynamicky zvyšovat geometrickou složitost modelů v reálném čase, což je zásadní pro vytváření vysoce detailních a vizuálně působivých scén bez nutnosti velkého množství polygonů uložených v paměti.

Tento druh shaderů pracuje ve třech hlavních fázích: tessellation control shader (TCS), tessellation primitive generator (TPG) a tessellation evaluation shader (TES).

- TCS – umožňuje vývojářům určit, jak se má geometrie dělit a jak detailní má být
- TPG – na základě TCS generuje nové vrcholy a definuje primitiva
- TES – vypočítává konečné pozice vrcholů na základě primitiv z TPG

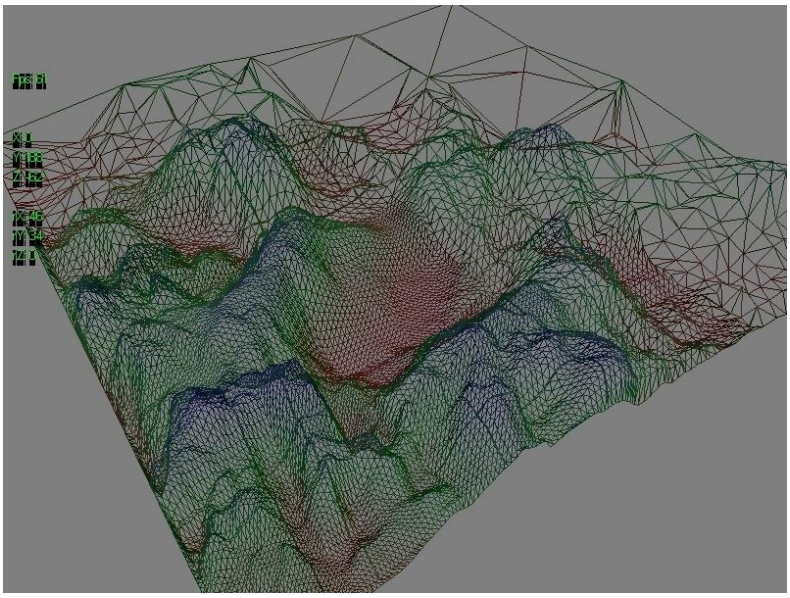
Výsledkem je geometrie, která může být mnohem detailnější v oblastech blízko pozorovatele a méně detailní ve vzdálenějších oblastech. To lze využít jak při generování terénu, tak dalších objektů ve scéně a zároveň je to velice šetrné na výkon celé aplikace.

Přestože jsou tessellation shadery velmi výkonným nástrojem, je důležité brát v úvahu jejich výpočetní náročnost. Vysoce detailní tessellace může zpomalit výkon, pokud není správně implementována anebo je použita nadměrně. Proto je důležité najít správnou rovnováhu mezi detailností terénu a výkonem, aby bylo zajištěno, že aplikace bude běžet hladce a svižně, pokud možno na všech hardwarových prostředcích, na kterých aplikace bude spuštěna.

Jako příklad pro představení výsledku si lze představit terén, který je různorodý nebo hornatý, stejně jako na obrázku 3. Lze si všimnout, že terén, který

je blíže ke kameře, je složen z více trojúhelníků než terén, který je od kamery vzdálen více. Při pohledu na takto malý terén se tessellation shadery můžou zdát jako ne tolik důležité a podstatné, což pro tvorbu pouhého terénu, který není tolik členitý, či terénu, který je limitovaný, jako ten na obrázku, může být pravda, proto tento shader nebyl použit v praktické části této práce, avšak při tvorbě rozsáhlejších světů s objekty je potřeba šetřit dostupné prostředky co nejvíce, tudíž je dobré znát o jeho existenci, jelikož se v komerční sféře hojně využívá a prakticky neexistuje jakýkoliv větší a náročnější projekt bez tessellation shaderů.

V praktické části této práce se s těmito shadery avšak nepracuje z důvodu porovnávání algoritmů samotných a jejich výpočetních náročností bez použití shaderů, který by tento výpočetní výkon jakýmkoliv způsobem ovlivňovali, což v případě těchto shaderů by se stalo (vertex shadery a fragment shadery jsou na druhou stranu nutné k samotné existenci takovéto aplikace). Na druhou stranu, rozepsány jsou proto, protože při vývoji terénu za použití níže zmíněných algoritmů jsou tyto shadery v zásadě esenciální pro tvoření jakýkoliv složitějších aplikací.



**Obrázek 3 - Ukázka využití tessealce při tvoření terénu** [6]

### 2.3.4 Ostatní druhy shaderů

Další shadery, které mohou být vhodné pro procedurální generování terénu mohou býti shadery Geometrické a Compute shadery.

Geometrické shadery mohou být použity k procedurálnímu generování detailů na geometrii, například k vytvoření vegetace, srsti zvířat, nebo k procedurálnímu přidávání geometrických detailů na modely v reálném čase. Tyto shadery jsou tedy využity v pokročilém tvoření světa a přesahují potřeby této práce. Compute shadery mohou být zásadní díky své schopnosti provádět složité výpočty paralelně a používají se zejména pro negrafické výpočty. Jsou ideální pro simulace, jako jsou částicové systémy, simulace tekutin, nebo dokonce celé ekosystémy a procedurální generování terénů na základě komplexních pravidel, tedy opět možnosti těchto shaderů přesahují potřeby této práce.

## 2.4 Geometrická primitiva

Výhoda používání grafických rozhraní, jako je OpenGL, jsou právě již grafická primitiva, která vývojářům usnadňují práci při vytváření komplikovaných terénů a světů. Tyto primitiva dokážou vytvářet přímky, polygony a další různé tvary, ze kterých poté lze vytvářet komplikovanější tvary, jako jsou čtverce, kruhy či trojúhelníky.

Všechny objekty, všechen terén, které jsou v počítačových hrách nebo filmech vytvářených počítačovou grafikou, jsou tvořeny přes nejjednodušší a nejuniverzálnější tvar, který existuje – trojúhelník. Za pomocí trojúhelníku lze totiž vytvořit jakýkoliv obrazec a tvar, ať už pouhým skládáním nebo zmenšováním. Trojúhelníky se také často dělí na menší a menší trojúhelníky vně již existujících trojúhelníků pro dosáhnutí detailnějšího prostředí a tvarů. Tato metoda se využívá zejména pro vzdálené objekty od kamery, kde není zapotřebí vykreslovat celý detailní objekt, jelikož to je výpočetně náročné, a proto jsou zobrazeny pouze hlavní, velké trojúhelníky, a vnější, vedlejší, trojúhelníky jsou smazány. To udělá objekt sice méně detailní, ale naopak bude šetrný k výpočetní paměti a vůbec celé aplikaci.
V OpenGL a teorii o počítačové grafice jsou tři hlavní metody skládání trojúhelníků, prostý GL_TRIANGLES, tedy jednotné trojúhelníky, poté GL_TRIANGLE_STRIP a dále GL_TRIANGLE_FAN

- GL_TRIANGLES – Tato metoda využívá samotné trojúhelníky, kde každé tři vrcholy definují jeden trojúhelník. Považuje se za nejuniverzálnější a

nejjednodušší metodu pro vykreslení trojúhelníku, která sice poskytuje největší kontrolu nad každým jednotlivým trojúhelníkem, což se může hodit při tvorbě detailních a složitých struktur, ale zato je velmi neefektivních z hlediska počtu vrcholů a přenosu dat.

- GL_TRIANGLE_STRIP – Metoda triangle strip velmi efektivně řeší problém, který metoda GL_TRIANGLES má, a to ten, že drasticky redukuje počet indexů tím, že každý trojúhelník sdílí vrcholy mezi sousedícími trojúhelníky. Po definování prvních dvou vrcholů každý další vrchol vytvoří nový trojúhelník společně s posledními dvěma vrcholy. Tato metoda je ideální pro tvorbu terénu, avšak pro tvorbu komplexnějších objektů a struktur je tato metoda nevhodná.
- GL_TRIANGLE_FAN – Triangle fan si lze představit, jak již název napovídá, jako vějíř, tedy první vrchol a zároveň index v posloupnosti je středovým vrcholem, a poté každý další pár vrcholů v prostoru tvoří trojúhelník s tímto středovým vrcholem. Použití této metody je hlavně u vykreslování kruhů, konvexních polygonů nebo dalších tvarů, kde lze efektivně využít sdílení středového bodu. Jak již představivost a popis napovídá, tato metoda je nejméně efektivní, dokonce lze říci, že je absolutně nevhodná pro generování terénu.

## 2.5 Alternativní grafické API

OpenGL ovšem není to jediné grafické rozhraní, které se v dnešní době používá. Ve skutečnosti to není ani jedno z hlavních rozhraní, které se používá. Velká většina vývojářů a moderních firem používají pro své projekty, jako grafické rozhraní, poměrně nový Vulkan, vytvořený firmou Khronos Group, anebo již léta osvědčený DirectX, vytvořen společností Microsoft, protože oproti těmto dvou API je OpenGL již zastaralé. Na rozdíl od jeho novějších konkurentů, OpenGL je jednodušší na použití, což z něj dělá vhodného kandidáta pro potřeby jednotlivců či práci na malém projektu.

### 2.5.1 DirectX

DirectX v jeho nejnovější verzi DirectX 12 (k roku 2024) je preferovaný pro svůj výkon na tvorbu počítačových her kvůli své schopnosti maximalizovat výkon hardwaru, se kterým pracuje. Jeho již zmíněna verze 12, která spatřila světlo světa již v roce 2015, má velice rozmanité vlastnosti oproti OpenGL, například spravuje paměť mnohem lépe, což je zásadní pro vytváření výkonově náročnějších her. Taktéž dokáže pracovat s vícero CPU jádry, co taktéž zvyšuje náročnost rozhraní a vizuálně zlepšuje výsledek. I když jsou metody, jak rozdělovat práci mezi vícero jádry v rozhraní OpenGL, OpenGL jako takové to samo o sobě neumí, na rozdíl od DirectX či Vulcanu. Nevýhoda a limitace DirectX je ovšem jeho rozmanitost, co se týče platform, na kterých působí. Ten může působit pouze na platformách od Microsoftu, tedy v ekosystémech Windows a Xbox.

### 2.5.2 Vulcan

Vulcan je oproti DirectX nízko úrovňové multiplatformní API, dodává vývojářům vyšší kontrolu nad grafickým hardwarem a je více optimalizováno pro práci s vícejádrovými procesory k použití vícero jader. Další vlastnost, kterou se Vulcan liší od ostatních, je že má explicitní správu paměti, na rozdíl od OpenGL a DirectX, které paměť abstrahují. To znamená, že Vulcan po vývojáři vyžaduje, aby přímo spravoval alokaci a dealokaci paměti a díky tomu může vést k lepšímu a efektivnějšímu využití všech hardwarových možností. To je zároveň do určité míry i nevýhoda, jelikož tato metoda je složitější, než mají ostatní API a pro nezkušené vývojáře to znamená nemalé komplikace v průběhu vývoje.

# 3 Procedurální generování v počítačové grafice

Předtím, než dojde k prozkoumání metody procedurálního generování a již určitých algoritmů, je nezbytné vysvětlit, co metodika procedurálního generování je a jaké má vlastnosti.

## 3.1 Představení problému

Při uslyšení pojmu procedurální generování si většina lidí představí již zmíněné videohry, nejčastěji nekonečné. Nekonečný terén, nekonečné herní úrovně, nekonečné světy. „Procedurální generování jsou dvě velká slova pro jednu jednoduchou věc: tvorba dat počítačem“[7] tedy pro uvedení do kontextu, světy vytváří počítač automaticky, a ne vývojáři ručně. Ti počítači naopak zadají sadu pravidel a metod, jak danou problematiku vyřešit a program podle těchto pravidel vytvoří zcela unikátní a náhodný svět, objekty i nástroje. Je důležité si uvědomit, že procedurální se nerovná náhodné.

### 3.1.1 Náhodné generování

Náhodné generování se zakládá na náhodných, ale předem předdefinovaných procesech. Jako příklad je možné si představit nějakou počítačovou hru, kde bude za cíl dojít z místnosti A do místnosti B, a mezi těmito místnostmi budou další místnosti s nepřáteli. V náhodném generování budou všechny místnosti, od designu až po nepřátele již předem definované, navrhnuté a nadesignované. Algoritmus náhodného generování poté vezme startovací místnost, kde hráč začíná, a náhodně pokládá další místnosti z již předem vytvořených. To znamená, že pokud vývojář vytvořil ručně 20 místností a algoritmu nastavil pravidlo, že v jedné hře jich může být celkem 7, tak pravděpodobnost, že se hráč narazí na identickou místnost je poměrně vysoká.

**Obrázek 4 - Ukázka předtvořených místností ze hry Binding of Issac [vlastní zpracování]**

### 3.1.2 Procedurální generování

Procedurální generování je do určité míry podobné. Také se řídí předdefinovanými pravidly, avšak nevyužívá žádný obsah vytvořený vývojářem. Pokud by se mělo pokračovat v příkladu z předešlé kapitoly, všechny místnosti by byli počítačem vytvořené, tudíž by zaručovaly téměř neopakovatelný zážitek a každý průchod hrou by byl v něčem jiný.

Hlavním rozdílem, mezi procedurálním generováním a náhodným generováním, je ten, že při použití procedurálního generování, s výjimkou Wave Collapse algoritmu, může být zajištěna opakovatelnost, a to pomocí tzv. seedu.
Seed je výsledkem ať už nějakého algoritmu, který generuje pseudo-náhodná čísla, třeba sekvenci čísel z čísla π (Například tučně znázorněná čísla – 3,1415926**53589793238**4626433), nebo uživatelem zadané číslo, případně, pokud je algoritmus pro zpracování seedu schopný, i znaky.

Algoritmus, které toto číslo obdrží, ho poté zpracuje a vrátí seznam čísel, která budou pokaždé stejná, čímž dosáhneme stoprocentní opakovatelnosti generovaného obsahu, což u algoritmů náhodného generování nikdy dosáhnout nemůžeme.

# 4 Algoritmy procedurálního generování

Novodobým vývojářům jsou známy několik algoritmů pro procedurální generování terénu, avšak některé algoritmy jsou vhodnější než ty druhé, proto je důležité znát ty hlavní a nejpoužívanější.

## 4.1 Perlinův šum

„Dobrý generátor náhodných čísel produkuje čísla, která nemají žádný vztah mezi sebou a jsou od sebe takřka nerozeznatelné“[8].

Jedním z algoritmů, který je nejznámější a je nejčastěji používán, je Perlinův šum, vytvořen profesorem z New Yorské univerzity doktorem Kenem Perlinem. Doktor Perlin původně vytvořil svůj šum při tvorbě známého a původního filmu „Tron“ z roku 1982. Ve filmu algoritmus použil při tvorbě počítačových vizuálních efektů, které se rozhodl procedurálně generovat. Perlinovi se algoritmus tak zalíbil, že ho po vydání filmu začal upravovat a vylepšovat až za již upravenou verzi v roce 1997 vyhrál cenu Academy Award.

V dnešní moderní době má Perlinův šum široké uplatnění zejména ve vytváření terénů a simulaci přírodních jevů. Jeho schopnost teoreticky podporovat různorodé aplikace je sice rozsáhlá, nicméně nejčastěji je využíván k vytvoření koherentních a vizuálně soudržných prostředí.

### 4.1.1 Kosinova interpolace

Perlinův šum využívá pro svůj terén zejména Kosínovu interpolaci, jelikož je nutné mít terén hladký a přirozeně zakřivený, jako tomu není u Lineární interpolace. Pro vytvoření Kosínovy interpolace se pouze lehce upraví původní vzorec pro výpočet Lineární interpolace a bude pojmenován jako „smoothLerp“.

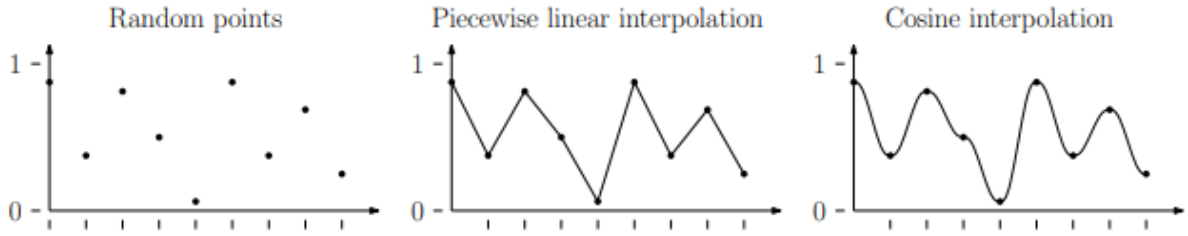
$$\text{smoothLerp}(a,b,t)=a+1/2(1-\cos(t\pi))\cdot(b-a)$$

**Vzorec 1 - Kosínova interpolace**

Kde:

- a = Počáteční hodnota
- b = Koncová hodnota
- t = Parametr interpolace

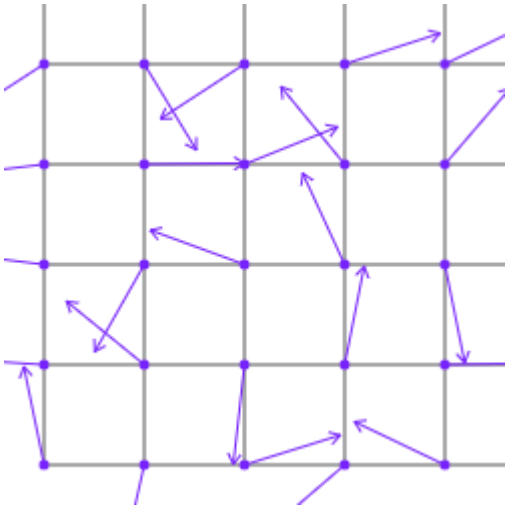
Tato triviální změna zajistí plynulejší změny mezi body a oku se čára bude zdát přirozená.




**Obrázek 5 - Ukázka interpolace [9]**

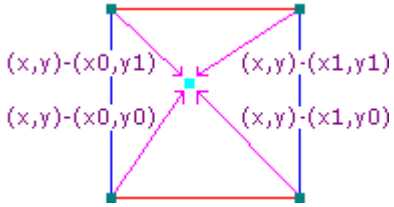
### 4.1.2 Sestrojení Perlinova šumu

Na začátku je náhodně vygenerováno 2D pole s body v mřížce o velikosti N x M. Pro každý bod v tomto poli poté vytvoříme náhodně vygenerovaný gradientní vektor, jejichž směr je náhodný, avšak jejich směr může být určen i určitými pravidly pro další zajímavé výsledky.



**Obrázek 6 - Vektory v Perlinově šumu [10]**

Pro zjištění barvy pixelu vytvoříme mezi 4 body z mřížky bod, zaznačený modrou barvou, a z každého okolního bodu, vyznačeny zelenou barvou, k nim jsou spočítány další 4, vzdálenostní, vektory.




**Obrázek 7 - Středový pixel [11]**

V momentě, kdy je znám pixel, který pro který je potřeba znát barvu, je spočítán skalární součin mezi vzdálenostním a gradientním vektorem. Tato operace funguje na základě faktu, že skalární součin dvou vektorů je roven kosinu úhlu mezi těmito vektory, vynásobeného magnitudami těchto vektorů.

$$dot(vec1,vec2)=cos(angle(vec1,vec2))\times vec1.length\times vec2.length$$

**Vzorec 2 - Vzorec pro nalezení středového bodu**

„Tímto se dosáhne toho, že pokud oba vektory jsou orientovány ve stejném směru, bude skalární součin roven“[11]:

$$1\times vec1.length\times vec2.length$$

**Vzorec 3 - Vzorec, čemu má být skalární součin roven, pokud jsou vektory orientovány ve stejném směru**

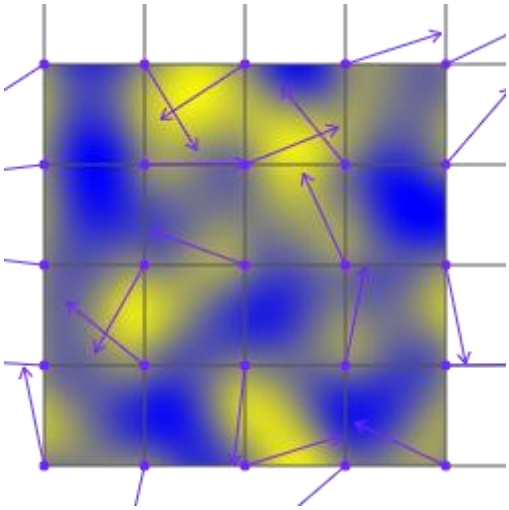
„A pokud jsou oba vektory orientovány v opačném směru, bude skalární součin roven“[11]:

$$-1\times vec1.length\times vec2.length$$

**Vzorec 4 - Vzorec, čemu mý být skalární součin roven, pokud jsou vektory orientovány v opačném směru**

Poslední možností je, že vektory budou na sebe vzájemně kolmé, v tomto případě by skalární součin byl roven 0.

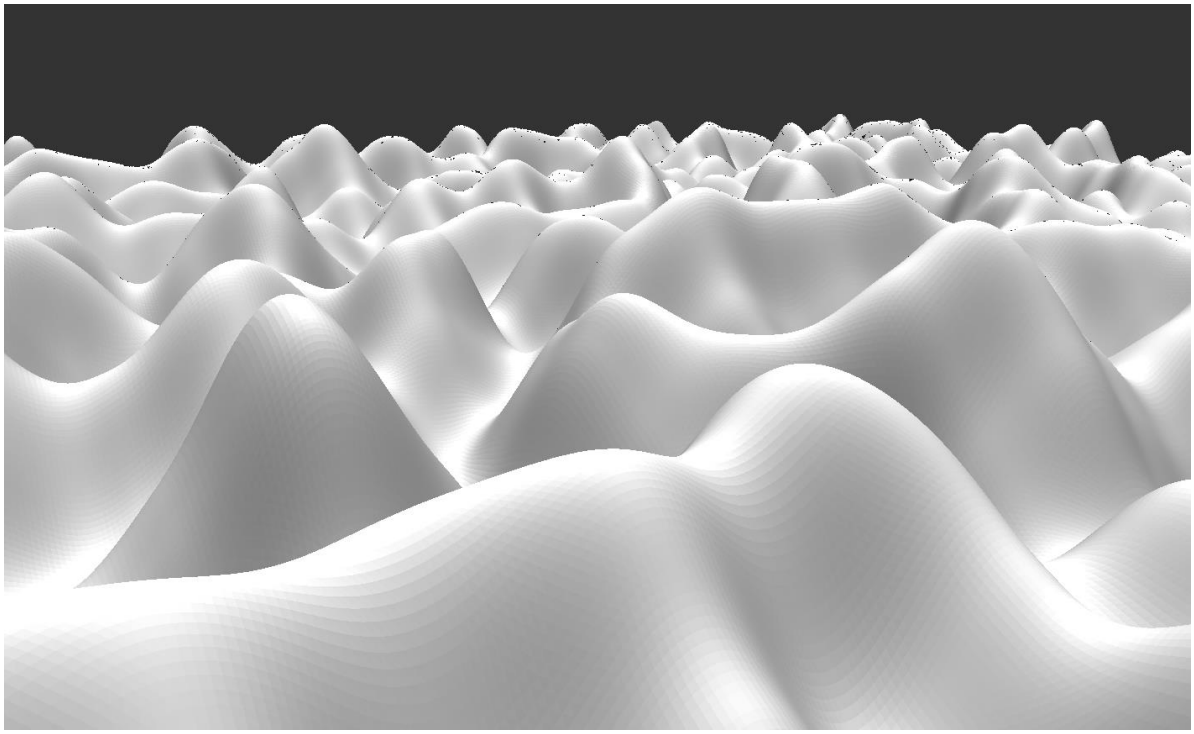
„Pokud výsledek skalárního součinu bude ve směru gradientu, bude kladný, pokud ovšem bude směřovat opačným směrem, bude výsledek záporný“[11]. Pomocí této metody lze určit barvu jednotlivých pixelů a výsledek celého algoritmu.



**Obrázek 8 - Vizualizovaný Perlinův šum [11]**

Hodnoty pozitivní a negativní se převedou na čísla, kdy pozitivní bude 0 (na obrázku 8. žlutá barva) a negativní 1, nebo naopak. Není ovšem potřeba mít tyto hodnoty, nula a jedna, vždy stejné. V některých v odborných článcích lze narazit na rozmezí od mínus jedné až do jedné, to je také možné, avšak toto rozmezí je již limitní, to znamená, že cokoliv jiného než <-1, 1> anebo <0, 1> nelze sestrojit, pokud se neupraví algoritmus, který generuje tento šum. Interval <0,1> byl zvolen z důvodu osobní preference. Pomocí těchto čísel lze poté sestrojit výškovou mapu s body, kde každý bod tedy nese hodnotu od, která je důležitá pro sestrojení realisticky vypadajícího terénu.

Za pomocí vyžití triangle stripu a jednoduchého algoritmu na spojení těchto bodů lze vytvořit vcelku uvěřitelný reálný terén. V této práci je ale za cílem vytvořit nekonečně velký terén za využití Perlinova šumu, proto je potřeba tento jednoduchý algoritmus trochu zkomplikovat, ten je vysvětlen v následující kapitole Segmentace, spolu s ukázkami implementace.



**Obrázek 9 - Implementace Perlinova šumu [vlastní zpracování]**

### 4.1.3 Využití knihovny FastNoiseLite

Při vytváření takto složitých algoritmů je dobré využít knihoven, které práci na procedurálním terénu zlehčují, avšak nedělají ji za kompletně za nás. Pro tuto práci byla využita knihovna FastNoiseLite, která sloužila jako prostředník mezi vstupem a výstupem, kde nám byl jako finální výstup poskytnuta výšková mapa.

## 4.2 Diamond-Square Algoritmus

Při tvorbě realistických scén a světů se vývojáři potýkají s problémem, jak a čím vytvořit svět, aby vypadal jeho struktury vypadali co nejvíce uvěřitelně a realisticky. Při aplikování Perlinova šumu ovšem nelze počítat s jeho reálností a věrohodností. Ačkoliv za určitých podmínek a různých modifikací Perlinova šumu je možné vytvořit reálně vypadající terén, lze nastavit různá prostředí, textury, vodní hladiny, je to většinou práce navíc, jak programátorská, tak i grafická, a proto je potřeba využít, nebo alespoň zkombinovat již vytvořený terén s dalším algoritmem, a tím je právě Diamond-Square algoritmus.

Tento algoritmus je velice podobný algoritmu Midpoint Displacement. Někdy jsou tyto dva algoritmy dokonce považované za jeden, ačkoliv má Diamond-Square

algoritmus o pár kroků více než Midpoint Displacement algoritmus, který mimo jiné vznikl originálně pouze pro jednu dimenzi, načež vznikl právě Diamond-Square, který přidal o dimenzi navíc, a díky tomu představuje efektivní a poměrně jednoduchý pohled a přístup ke generování fraktálního terénu, který dokáže vypadat realisticky, uvěřitelně a je topograficky proměnlivý a korektní. To znamená, že výsledný terén nevypadá tak hladce, je více hrubý a více proměnlivý. Pomocí Diamond-Square algoritmu lze vytvářet poměrně rychle a efektivně generovat hory, údolí, rokliny i jeskyně a další, lidem známé terénních struktury a formace, s dostatečnou mírou náhodnosti, a přitom zachovávají určitou předvídatelnost ve své struktuře.

Diamond-Square algoritmus není využíván pouze v herním průmyslu, ale také ve vojenských či záchranářských simulací, nebo také slouží pro vizualizaci v geografických informačních systémech, známé pod zkratkou GIS, a výjimečně v architektuře.

### 4.2.1 Implementace v teorii

Implementace algoritmu je ve své podstatě velmi jednoduchá. Jedna z výhod tohoto algoritmu je, že není zapotřebí si pamatovat postup, jelikož postup vyplívá přímo z názvu algoritmu, tedy Diamond a Square. Toto budou dva důležité kroky, které k vytváření terénu za pomocí Diamond-Square algoritmu jsou zapotřebí.
Celý princip spočívá v tom, že za pomocí 4 náhodně vygenerovaných, či určených, hodnot rohů, ať už, v tomto případě, segmentu, popřípadě jakékoliv jiné struktury, ve dvou dimenzionálním listu hodnot budou sestaveny a vypočítány zbytky hodnot uprostřed tohoto segmentu.

Představit si to lze jako čtverec, který bude rozdělen na mřížku, například, 5x5, a budou známy hodnoty rohových buněk. Úkolem algoritmu je tedy vypočítat ostatní čísla vně tohoto čtverce.

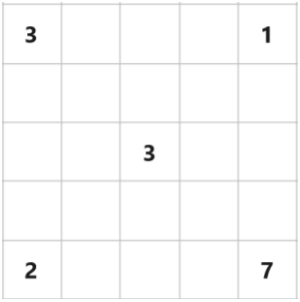
Jedna velká nevýhoda tohoto způsobu generování terénu, že je potřebné myslet na velikost terénu neboli 2D pole hodnot, což je jedna z informací, která je předávána algoritmu k vygenerování terénu. Ten musí mít velikost $\mathbf{2^n + 1}$ na výšku i na šířku, tedy velikosti terénu s čísly jako jsou 5x5, 17x17, 33x33. Tato podmínka

tedy výrazně limituje možnosti generování a pro případné kombinace s jinými algoritmy může způsobovat nemalé komplikace.

#### 4.2.1.1 Krok Square

Generování terénu začíná krokem square, tedy v českém jazyce čtverec, a je velice logický a intuitivní. V některých literaturách a článcích se tento krok nazývá jako Diamond, což je v této práci krok následující, avšak pojmenování tohoto kroku Diamond se zdá být nelogické a nesprávné, proto jsou tyto dva kroky obráceně pojmenované v této práci. V prvé řadě je potřeba vypočítat prostřední hodnotu tohoto čtverce. To se docílí vypočítáním průměru všech čtyř hodnot rohů tohoto čtverce, tedy jedinými počátečními hodnotami, které jsou na začátku známy. Ty jsou většinou, jak již bylo zmíněno, náhodné, ale pro účely této práce jsou ručně nastavitelné.

Pro ukázku výpočtů byli vybrány následující čísla rohů: 1,3,2,7. Při dosazení do rovnice a následném vypočítání, tedy výpočet průměru z těchto čtyř čísel, vychází číslo 3,25. Při výpočtu bodů v mřížce u tohoto algoritmu platí pravidlo, že neúplná čísla, tedy čísla, které mají hodnoty za desetinnou čárkou, jsou zaokrouhlena k nejbližšímu celému číslu. Proto je výsledek kroku square, tedy prostřední pole ve čtverci, číslo 3.




**Obrázek 10 - Krok square [vlastní zpracování]**

#### 4.2.1.2 Krok Diamond

Znalost prostřední hodnoty čtverce odemyká další krok pro výpočet zbylých neznámých hodnot v poli, který se nazývá Diamond. Stejně jako v předešlém kroku, kroku Square, je výpočet dalších čísel velice intuitivní a taktéž se bude jednat o výpočet průměru.

Zatímco předešlý krok počítal průměr z hodnot, které byly rozmístěné do čtverce, diamantový krok se bude tedy počítat do tvaru kosočtverce. Pro příklad se bude pokračovat ve výpočtu hodnot 5x5 mřížce, které byli zadané v kroku Square. Na rozdíl od předešlého kroku, v tomto případě nelze určit průměr ze čtyř hodnot, jako tomu bylo v kroku Square, jelikož do tvaru kosočtverce v mřížce o velikosti 5x5 nelze nalézt 4 hodnoty. Proto je tedy nutné využít lidskou představivost a ten čtvrtý bod si domyslet za hranicí mřížky, avšak do výpočtu tento bod přidán nebude. Výpočet tedy pro první kosočtverec bude průměr ze tří hodnot, 3,3 a 2. Výsledný průměr těchto čísel je 2,66 a tedy stejně, jako u předchozího kroku, se číslo zaokrouhlí na číslo 3, tedy nejbližší celé číslo, které se do mřížky terénu přidá doprostřed již představeného celého kosočtverce.

| | | | | |
|---|---|---|---|---|
| 3 | | | | 1 |
| | | | | |
| 3 | | 3 | | |
| | | | | |
| 2 | | | | 7 |

**Obrázek 11 - Krok diamond [vlastní zpracování]**

Tímto způsobem se dopočítají všechny zbývající kosočtverce, tedy tento způsob v mřížce 5x5 se provede ještě třikrát.

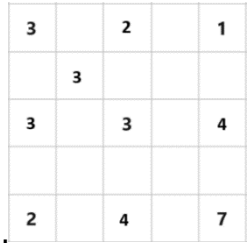
| | | | | |
|---|---|---|---|---|
| 3 | | 2 | | 1 |
| | | | | |
| 3 | | 3 | | 4 |
| | | | | |
| 2 | | 4 | | 7 |

**Obrázek 12 - Jedna iterace Diamond – Square algoritmu [vlastní zpracování]**

#### 4.2.1.3 Opakující se postup

V momentě, co se vypočítá nejdříve Square krok a poté Diamond krok, lze celý tento proces opakovat. Lze si všimnout v Obr.12., že po vypočítání kroku Diamond, vznikne další příležitost dopočítávat hodnoty pomocí Square kroku, tedy

další číslo, které by bylo dopočítáno, by byl průměr hodnot čtverce s hodnotami 3,2,3 a 3, jejichž průměr by byl po zaokrouhlení 3 a dosazen bude opět do středu tohoto malého čtverce.




**Obrázek 13 - Pokračování algoritmu Diamond-Square [vlastní zpracování]**

Tyto dva způsoby se budou opakovat, zpravidla se vždy střídají, nikdy nelze použít dva stejné kroky za sebou, dokud se nevyplní celá mřížka o velikosti 5x5. Výsledná doplněná mřížka poté udává body, které jsou potřeba k sestrojení terénu, takzvané height pointy, tedy výškové body. Tyto body nesou informaci, podobně, jako u Perlinova šumu, jak vysoko bod v terénu má být a po následném spojením patřičné techniky, například GL_TRIANGLE_STRIP, vytvoří již výsledný terén.

### 4.2.2 Implementace v praxi

Algoritmus Diamond-Square se v programovacím prostředí skládá z několika triviálních funkcí, které následují stejný postup, jako při vysvětlování teorie.

#### 4.2.2.1 Funkce generate

Tato funkce zahajuje a řídí celý proces generování. Začíná s velikostí kroku, která se postupně snižuje, což odpovídá zvyšující se detailnosti terénu spolu s každou iterací. Algoritmus se skládá ze dvou kroků, diamond a square, které se opakují v různých měřítkách (*step* a *halfstep*)

```
public void generate() {
    int step = size - 1;
    for (int halfStep = step / 2; halfStep > 0; halfStep /= 2, step
/= 2) {
        for (int y = halfStep; y < size - 1; y += step) {
            for (int x = halfStep; x < size - 1; x += step) {
                squareStep(x, y, halfStep);
            }
        }
        for (int y = 0; y < size; y += halfStep) {
            for (int x = (y + halfStep) % step; x < size; x += step)
{
                diamondStep(x, y, halfStep);
            }
        }
    }
}
```

**Kód 1 - Funkce generate**

#### 4.2.2.2 Funkce squareStep

V této funkci je realizován krok Square. Ten spočítá průměr z dostupných (maximálně čtyř) sousedních středových bodů a okrajových bodů z předchozích iterací (pokud se nejedná o začátek algoritmu), k nimž přidá náhodný posun, v kódu pod atributem „*offset*“, což napomáhá vytvořit realističtější a vizuálně zajímavější terén.

```
private void squareStep(int x, int y, int halfStep) {
    float avg = (heightMap[x - halfStep][y - halfStep] + heightMap[x
+ halfStep][y - halfStep] +
            heightMap[x - halfStep][y + halfStep] + heightMap[x +
halfStep][y + halfStep]) * 0.25f;
    float offset = useRandomness ? randValue(halfStep) : 0;
    heightMap[x][y] = avg + offset;
}
```

**Kód 2 - Funkce squareStep**

#### 4.2.2.3 Funkce diamondStep

Tato funkce aplikuje krok Diamond, což znamená, že spočítá průměrnou hodnotu z čtyř vrcholů tvořících diamant (čtverec otočený o 45 stupňů) a přidá k této průměrné hodnotě náhodný posun, v ukázce kódu opět atribut „offset“, pro simulaci nerovnosti terénu.

```
private void diamondStep(int x, int y, int halfStep) {
    float avg = 0;
    int n = 0;
    if (x - halfStep >= 0) { avg += heightMap[x - halfStep][y]; n++;
}
    if (x + halfStep < size) { avg += heightMap[x + halfStep][y];
n++; }
    if (y - halfStep >= 0) { avg += heightMap[x][y - halfStep]; n++;
}
    if (y + halfStep < size) { avg += heightMap[x][y + halfStep];
n++; }

    avg /= n;
    float offset = useRandomness ? randValue(halfStep) : 0;
    heightMap[x][y] = avg + offset;
}
```

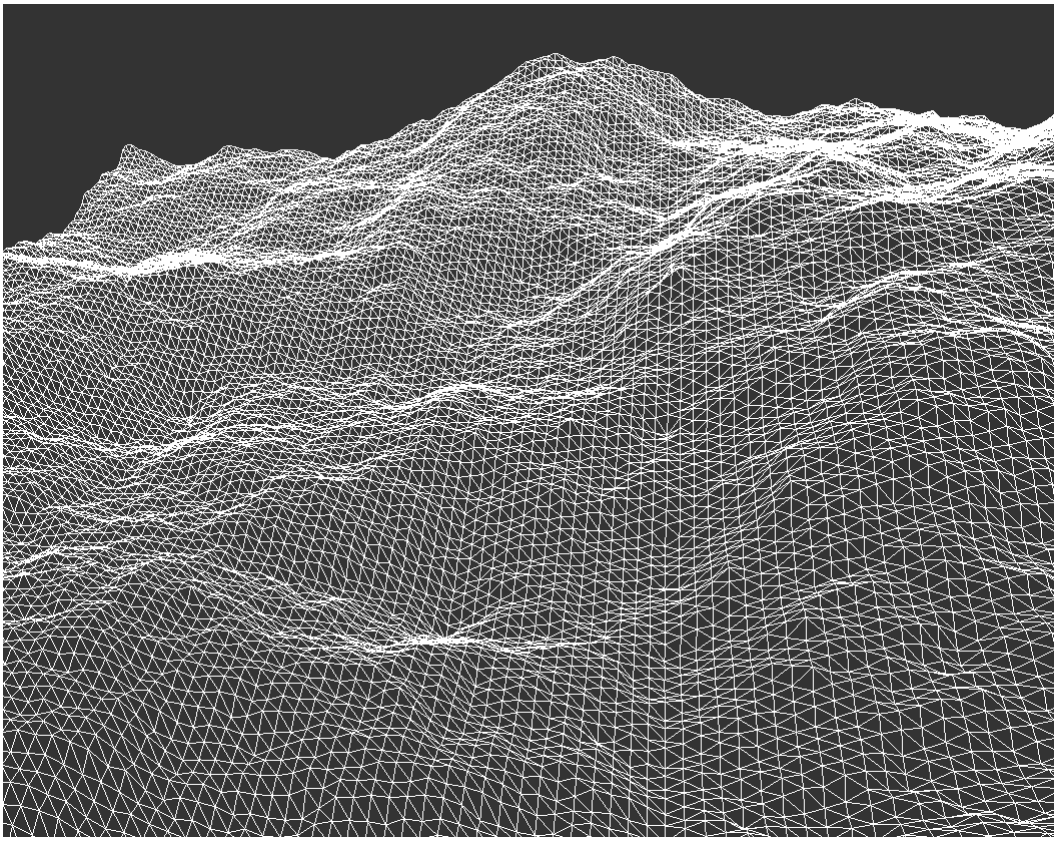
**Kód 3 - Funkce diamondStep**

### 4.2.3 Offset

Algoritmus sám od sebe bez žádného přídavného offsetu, avšak generuje terén příliš plochý na to, aby vůbec spadal do definice samotného terénu. Z tohoto důvodu je důležité k tomuto algoritmu přidat nějaký offset.

Pro zachování pravidelnosti za pomocí seedu je doporučeno používat nějakou funkci, která tento seed bude zpracovávat. Tato funkce může být napsána vlastním způsobem, avšak doporučuje se tento algoritmus kombinovat třeba s Perlinovým šumem, nebo jiným algoritmem.

Z důvodu zachování určité variace namísto pravidelnosti byl ovšem zvolen naprosto přístup naprosto náhodných čísel, tedy matematickou funkcí v základním balíčku Java pod názvem „java.util.Random“. Tato funkce je poté použita k vytvoření náhodného offsetu, který terénu dodá realističnost a uvěřitelnost, že se o terén opravdu jedná.



**Obrázek 14 - Implementace Diamond-Square algoritmu [vlastní zpracování]**

## 4.3 Wave Function Collapse

Další z algoritmů, který již není tak populární, jako předešlé dva algoritmy, je takzvaný Wave Function Collapse, zkráceně WFC, do češtiny je tento název možné přelozit jako „Kolaps vlnové funkce“, což opět perfektně popisuje způsob fungování algoritmu. Jde tedy o techniku založenou na principu kolapsu „kvantové“ [12] vlnové funkce, která se objevuje především v kvantové fyzice, byla adaptována pro digitální prostředí s cílem vytvářet pokročilou počítačovou grafiku.

Pro zajímavost byl nástroji umělé inteligence ChatGPT ve verzi 4 položen dotaz, v kterých odvětvích se Wave Collapse používá, načež jeho odpověď byla následující: „*Wave Function Collapse (WFC) algoritmus je fascinující nástroj v oblasti procedurální generace, který nachází uplatnění ve videohrách, generování umění, architektuře a mnoha dalších kreativních a technických oborech*.“. Následně nastal úkol tuto informaci buď potvrdit nebo vyvrátit. Každý, kdo je zainteresovaný do programování her a vytváření terénu ví, že WFC je především používán pro generování terénu, hlavně ve dvoudimenzionální scéně, kde se používá k tvorbě

často nekonečného terénu, občasně i ve trojdimenzionální scéně, kde je kvůli komplexitě a výkonné náročnosti takřka nepoužívaný k tvorbě nekonečných světů, tedy alespoň během výzkumu žádný takovýto projekt nebyl nalezen a ani není obecně znám. Tudíž tuto jednu informaci lze instantně bez rozmýšlení potvrdit.
Na začátek je nutné podotknout, že na téma Wave Function Collapse v počítačové grafice neexistuje mnoho odborné literatury, jelikož je zřídka používaný, komerčně takřka vůbec kvůli jeho komplexitě a zároveň určitým limitacím.

Při průzkumu tohoto tématu byl nalezen jeden článek, který výslovně říká následující, přeloženo do českého jazyka: „Je (Wave Function Collapse) především používán pro vytváření obrázků, ale také je schopný vytvářet budovy, města a skate parky.“ [13].

Tímto tedy lze potvrdit tvrzení některá tvrzení inteligence potvrdit, tedy konkrétně myšlenku o vytváření umění. Vytváření architektonických struktur pomocí WFC, se znalostí, jak tento algoritmus funguje, o čemž hovoří následující podkapitola, je velice obtížné si představit.

### 4.3.1 Implementace v teorii

Jak již bylo naznačeno, WFC je určen i k vytváření obrázků, a právě na obrázcích, či již předdefinovaných a předurčených struktur, nebo teoreticky i segmentů, které můžou být různých velikostí, či se můžou lišit metodou generování, je celý tento algoritmus postavený.

Pro začátek je potřeba tedy mít nějaký list struktur, obrázků, v podstatě čehokoliv, z čeho bude výsledný terén „poskládán“. Ve článku, který byl citován výše, přiřazují tuto strukturu jako k plánování svatby, avšak je dobré si celý problém představit i na jiných příkladech, jelikož každý smýšlí jinak a je možné, že to bude více pochopitelné.

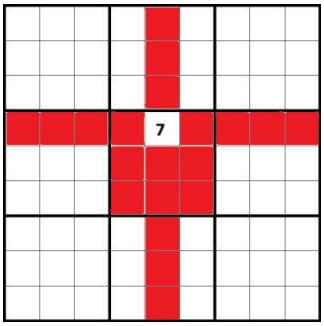
### 4.3.2 Podobnost se Sudoku

Sudoku, hra z dětství mnoha lidí, je hra čísel, která je hrána v původní verzi v 9x9 mřížce, kde jsou všechny tyto 81 buňky rozděleny do svých 3x3 boxů, tedy celkový počet těchto boxů je 9. Úkolem této hry je doplnit do všech buněk číslo od jedné do devíti za určitých pravidel. Pravidla pro hraní Sudoku jsou triviální

k pochopení. V každém řádku, sloupečku a boxu se musí vyskytovat čísla od jedné do devíti a v každým jednom řádku, sloupečku a boxu se tyto čísla mohou vyskytovat pouze jednou, to znamená, že v každém řádku, sloupečku a boxu bude pouze jednou číslo jedna, pouze jednou číslo dva atd.

Pokud se nepoužije jako příklad již předpřipravená mřížka, ale začne se od bodu, kdy mřížka je prázdná, nastáván moment, kdy každá buňka je ve stavu, který je převzat z kvantové mechaniky, tzv. superpozice. Je to tedy moment, kdy každá jedna buňka může nést hodnotu jakékoliv z předem definovaných čísel, tedy v případě Sudoku od jedné do devíti.

V 99 % případech avšak Sudoku nezačíná „naprázdno“, ale některé buňky již většinou nesou nějakou hodnotu. Do mřížky je pro představení problému do buňky v prostředním boxu přidána hodnota sedm. Tato buňka tedy ztrácí superpozici a zároveň ovlivňuje ostatní buňka v téže boxu, řádku a sloupečku. Tyto ovlivněné buňky tedy taktéž ztrácejí titul superpozice, jelikož již nemůžou nést hodnotu již určené buňky, tedy hodnotu sedm. Tento jev lze tedy nazvat „kolaps“, proto Wave Collapse Function.




**Obrázek 15 - Logika sudoku WFC [vlastní zpracování]**

Při startu hry, jak již bylo zmíněno, tedy hráč začíná již do určité míry s předvyplněným hracím polem, mřížkou. Nastává tedy situace, kde každá buňka, která ještě nemá žádnou hodnotu, má určitý počet možností nebo hodnot, které jsou dostupné, aby splňovali předem nastavená pravidla a omezení. Hráč se v tomto případě dostane do situace, za předpokladu, že hru umí hrát a zná ten nejlepší možný postup, tak, jak tomu je u WFC algoritmu, kdy potřebuje zanalyzovat možnosti a najít buňku, které má nejmenší počet možných dostupných hodnot neboli také tzv. „entropií“. Důvod, proč je potřeba vybrat buňky s nejmenším počtem

entropií je velice logický, je tedy protože tím se snižují šance k chybě a udělání špatného rozhodnutí.

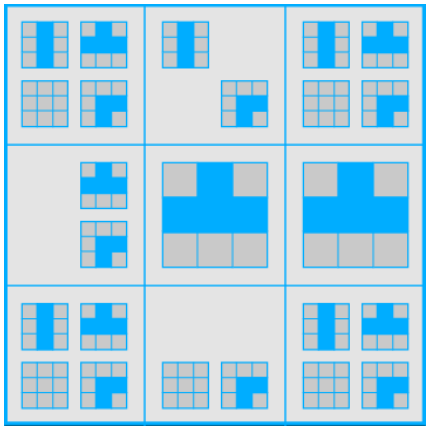
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

**Obrázek 16 - Ukázka sudoku [14]**

Tímto způsobem tedy hráč iteruje přes všechny možný, a ještě nevyplněný buňky, až se dostane do bodu, kdy všechny buňky budou vyplněné a za předpokladu, že nebyli porušeny žádná pravidla, hra končí.

### 4.3.3 Wave Function Collapse za použití Sudoku logiky

Při generování terénu v počítačové grafice pomocí Wave Collapse Function algoritmu, algoritmus na generování nahlíží úplně stejně, jako lidé na řešení Sudoku. Je zadán určitý grid, mřížka, kde všechny buňky na samém počátku budou mít vlastnost superpozice. Poté je vybrána počáteční buňka, buď náhodně, nebo také za nějaké podmínky, a je jí přiřazena nějaká hodnota, třeba „les“, tedy toto políčko bude ve finále políčko, kde bude například textura lesa. Následně algoritmus ví, že vedle buněk, co mají hodnotu „les“, můžou být pouze buňky s hodnotou „pole“, „hora“ a „dům“. To zredukuje sousedící buňky na tyto tři možnosti. Pokud budou v celé mřížce další předurčený pravidla, postupuje se stejně, jak to bylo vysvětleno u Sudoku, pokud ovšem je zadána pouze tato první počáteční buňka, a tedy algoritmus má na výběr ze tří možností, je jedna z těchto možností vybrána zcela náhodně.
Tímto způsobem, který připomíná způsob, kterým se pohybují vlny, proto Wave FC, se postupuje až do úplného zaplnění mřížky buňkami s unikátními hodnotami.



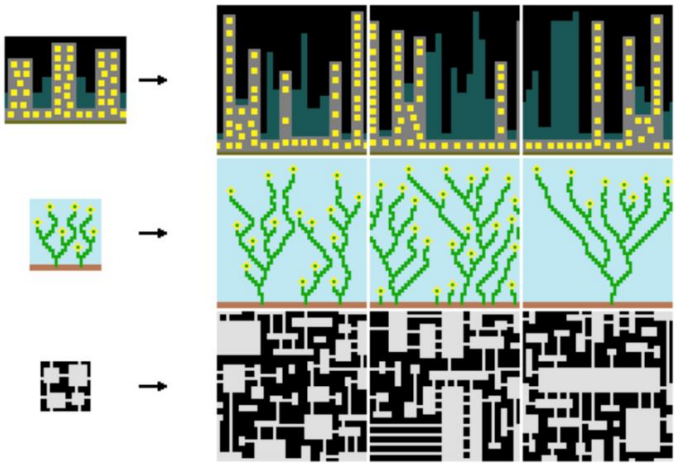
**Obrázek 17 - Možnosti při generování algoritmem WFC pro smyšlený terén [15]**

Krása tohoto algoritmu je, že i když algoritmus při generování vypadá hladce a bez zastavení, tak používá iterace a při umělém zpomalení generování, což u algoritmu WFC lze, na rozdíl od předešlých algoritmů, který výsledné segmenty vrací v celku, je možné pozorovat, jakým způsobem algoritmus pracuje, a pokud jsou vizuálně vyobrazeny, jako v obrázků 17, možnosti výběru hodnot, dokáže tento algoritmus být opravdu magický.

Po skončení všech iterací, tedy za všechny buňky mřížky jsou dosazeny hodnoty, algoritmus končí a dále již nepokračuje, tedy říkáme, že se algoritmus zhroutil, anglicky collapsed, tedy proto WF Collapse.

V praxi se obvykle při použití algoritmu WFC negeneruje terén prostřednictvím segmentů, ale již hned na začátku je celý terén vygenerován, to je taktéž zásadní nevýhoda tohoto algoritmu.

Další nevýhodou algoritmu je to, že způsob, jakým je strukturovaný, je velice „hranatý“, tedy jednotlivé části jsou tvořeny v mřížce, což dělá z terénu jakousi nerealistickou strukturu. To je také hlavní důvod, proč se algoritmus používá především v prostředí 2D a nikoliv 3D, ačkoliv jej lze použít ve 3D. Ve dvoudimenzionálním prostředí se tyto nerovnosti zanedbávají, jelikož není vidět výška terénu, zatímco v trojdimenzionálním prostředí je výška neboli osa Z, důležitým prvkem a bez této osy by terén neměl žádný smysl.



**Obrázek 18 - Výsledný obrázek za použití WFC [16]**

## 4.4 Voroného diagramy

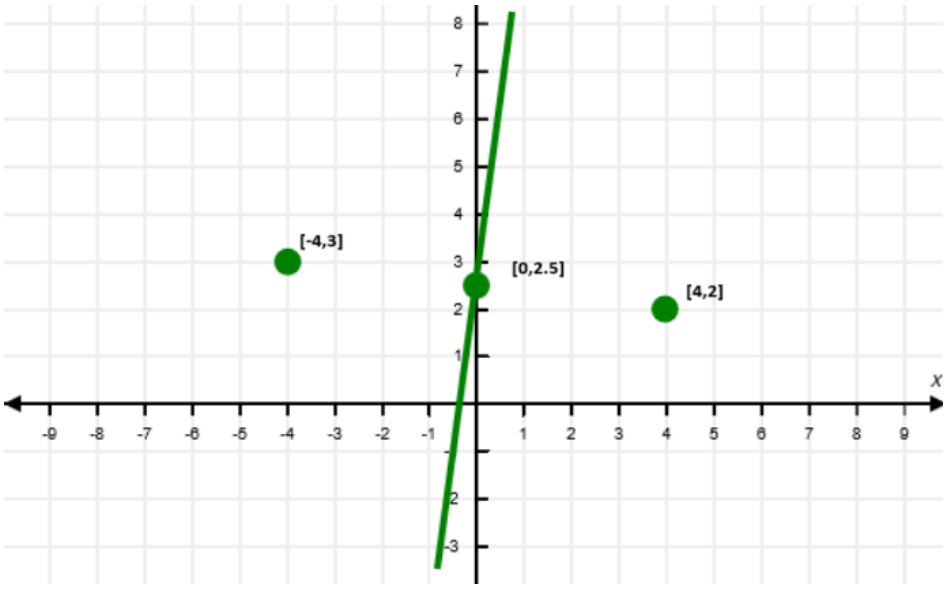
Další z algoritmů, v některých případech lze říct jevů, si našel fascinující uplatnění nejenom v počítačové grafice, ale také v reálném světě. Příkladem z reálného světa můžou být třeba „skvrny, které mají žirafy na svém těle, nebo také povrch či textura křídel vážek či dokonce se struktura stěn obyčejných listů považuje za Voroného diagram v přírodě“ [17]. V počítačové grafice se poté tyto algoritmy uplatňují ve 2D při tvorbě výtvarných obrazců nebo textur a ve 3D zas naopak při generování oblak či počasí, či detailnější povrch objektů a struktur, nebo také při vytváření biomů, tedy oblastí, které rozdělují virtuální svět na zóny se specifickými vlastnostmi, to může být třeba město, les, poušť.

Tyto diagramy, jež jsou pojmenovány po ruském matematikovi Georgije Voroného, jsou založeny na rozdělení prostoru na regiony zvané Voroného buňky. Každá buňka představuje oblast prostoru blíže k určitému bodu (nazývanému generátor nebo střed) než k jakémukoli jinému. Tato jednoduchá, avšak výkonná koncepce umožňuje Voroného diagramům modelovat širokou škálu přirozeně se vyskytujících struktur a jevů.

### 4.4.1 Sestavení diagramu

„Je dána rovina, ve které je zadán N počet náhodně rozmístěných bodů. V této rovině je dále potřeba nějako rozdělit tyto body od sebe“[18], rozdělit je na segmenty, části, oblasti. V tom jsou ideální Voroného diagramy, které dělají přesně toto.

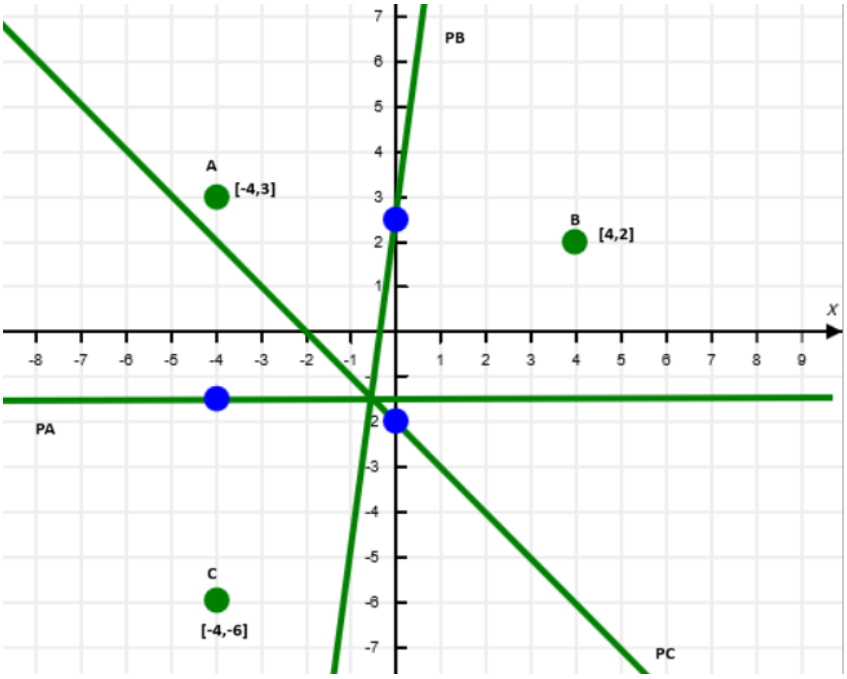
Pro pochopení, jak jsou tyto dva diagramy vytvářeny, budou zadány pouze dva body. Tyto body leží v rovině a jejichž souřadnice jsou [-4, 3] pro bod A, a pro druhý bod B [4,2]. Dalším krokem bude vytvořit mezi těmito body, nebo minimálně si představit, jakousi úsečku a vytvořit nový bod, který bude ležet uprostřed těchto dvou bodů. Po sléze se k této úsečce vytvoří kolmice, která bude protínat právě tento středový bod.




**Obrázek 19 - Kolmice procházejíc skrze středový bod [vlastní zpracování]**

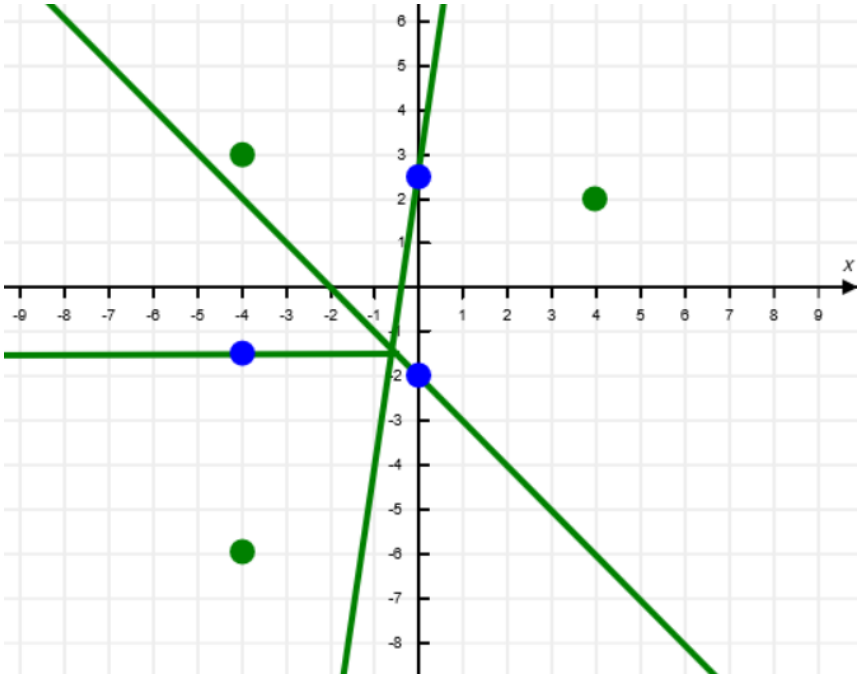
Tímto postupem byl v ukázce na Obr.19 vytvořen Voroného diagram. V tento moment, cokoliv, co bude ležet na levé straně od nově vytvořené kolmice bude blíže k bodu A, a naopak na pravé straně kolmice bude blíže k bodu B.

To je ovšem moc jednoduché a Voroného diagramy nikdy takto triviální nejsou. Pro zvýšení komplexity celého problému tedy bude přidán další, třetí, bod C, jež leží na souřadnicích [-4,-6]. Ze začátku se tedy bude postupovat úplně stejně, jako v případě se dvěma body. Jsou zadány tři body, tudíž budou vytvořeny tři středové body mezi jednotlivými body. Poté se opět vytvoří kolmice k pofiderní úsečce mezi jednotlivými body a zároveň bude protínat nově vytvořené středové body, ovšem toto nebude, jako tomu bylo v případě se dvěma body, finální krok.



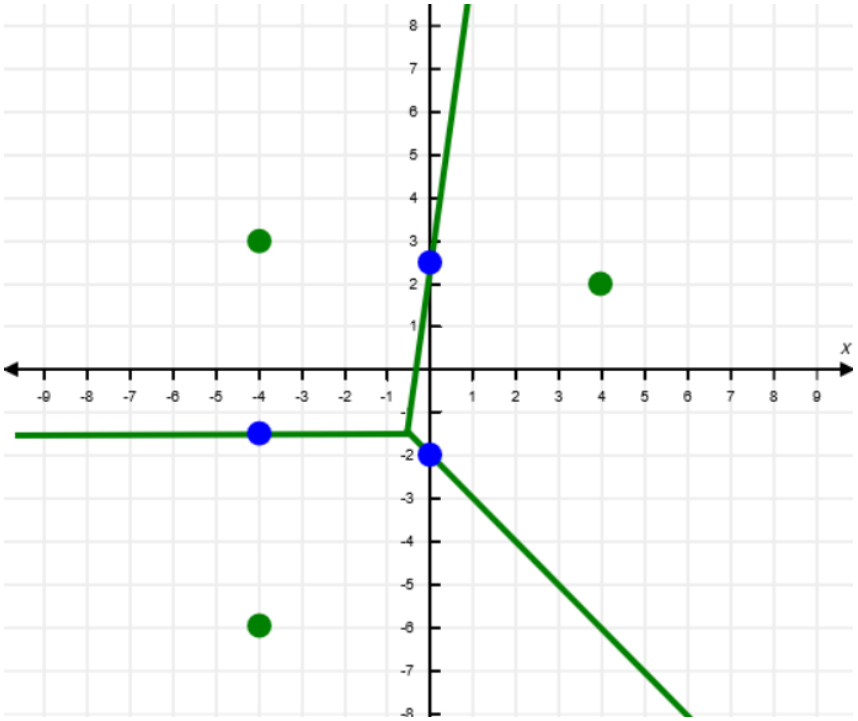

**Obrázek 20 - Výsledné rozdělení Voroného diagramů bez ořezání [vlastní zpracování]**

Na Obr.20 je možné si všimnout, že nyní, přímka PA, tedy nově vzniklá kolmice mezi body A a C, protíná přímky PB a PC. Zároveň je očividné, že všechny body na levé straně od průsečíků přímek na přímce PA jsou blíže bodům A a C a všechny body na přímce PA na pravé straně od průsečíků přímek jsou ovšem blíže k bodu B. Co je tedy za potřebí udělat, je odstranit pravou stranu přímky PA od průsečíků přímek PA, PB a PC. Tím tedy vznikne graf, jako na Obr.21




**Obrázek 21 Proces ořezání přímek v algoritmu Voroného diagramu [vlastní zpracování]**

Stejným způsobem se ořežou všechny přímky podlé stejné logiky, tedy jestliže všechny body na nově vzniklých kolmicích jsou blíže jinému bodu než toho, ze kterých je tvořen středový bod, který protíná kolmice, budou v bodě průsečíků všech tří kolmic zkráceny. Tímto postupem tedy vznikne tento výsledný graf a zároveň vzniknou chtěné Voroného diagramy.




**Obrázek 22- Výsledná podoba Voroného diagramu v teorii [vlastní zpracování]**

Tímto způsobem se tedy při tvoření diagramů iteruje přes všechny body v prostoru a dopočítávají se mezi nimi Voroného buňky. Výsledek je nejčastěji, jak již bylo zmíněno, používaný v kombinaci s dalšími algoritmy a slouží k rozčleňování terénu do zón, nebo pro tvorbu městských částí apod.

# 5 Segmentace terénu

V herním světě je velice častým požadavkem mít vygenerovaný terén nekonečný, aby si hráči měli nekonečné možnosti věcí, které mohou dělat a v ideálním případě hrát hru do konce časů. V tom velice pomáhá metoda segmentace terénu, v angličtině a ve světě hráčů je tato metoda známa spíše pod termínem „Chunks“. Jedná se o metodu, kdy díky skládání segmentů vyvolává pocit, že svět je doopravdy nekonečně veliký.

## 5.1 Segmenty

Jednotlivé segmenty si lze představit jako kostičky z puzzle. Jednotlivý dílek je vždy unikátní, má určité atributy, například specifická textura, velikost, a pasuje pouze k dalším 4 dílkům taktéž unikátním dílkům z celého světa, skládačky.

Tyto segmenty můžou být generovány z různých algoritmů, ať už Perlinova šumu, ze kterého jsou ukázky níže, nebo Diamond-Square algoritmu. Tyto algoritmy lze dokonce v jednom světě, pro vytváření komplexnějších světů a struktur, kombinovat, zde je však potřeba myslet na fakt, že tyto dva algoritmy musí na sebe nějakým způsobem navazovat. Například kombinace algoritmu Wave Collapse Function se bude obtížněji kombinovat s terénem vytvořeným pomocí Perlinovat šumu.

## 5.2 Segmenty za použití Perlinova šumu

V případě použití Perlinova šumu je za potřebí si stanovit, jak velké segmenty jsou zapotřebí či chtěné, dále je nutné určit škálu terénu, ta ovlivňuje osu X a Y a určuje, jak moc terén má být členitý, zdali bude spíše hladký či více kopcovitý, a poslední dva atributy každého segmentu je pozice X a Y v celém terénu.

Při určování velikosti je dobré si uvědomit, že by strany X,Y měli být stejné, tedy například 128x128 pixelů, 1024x1024 pixelů. Velikosti nerovných velikostí, například 128x300 pixelů není vhodné používat z důvodu navazování na segmenty, které budou dále modifikovány jiným algoritmem či dalšími nejrůznějšími atributy, aby terén ve finálním výsledku vypadal komplexněji a více reálně, avšak pro nejjednodušší případy a správně nastavený segmentační manager lze použít rozdílné velikosti X,Y.

Následně, za použití již vytvořeného algoritmu pro generování terénu, v tomto případě Perlinova šumu, použijeme jednoduchý algoritmus na vytvoření segmentu.

```
for (int y = 0; y < extendedSize; y++) {
    for (int x = 0; x < extendedSize; x++) {
        // Here, x and y represent the horizontal plane, with z
being the height
        float globalX = (chunkX * terrainSize + x) * terrainScale;
        float globalY = (chunkY * terrainSize + y) * terrainScale;
        float height = perlinNoise.getNoise(globalX, globalY) *
heightMultiplier;

        int vertexIndex = y * extendedSize + x;
        vertices[vertexIndex * 3] = globalX; // X - Front and Back
        vertices[vertexIndex * 3 + 1] = globalY; // Y - Left and
Right
        vertices[vertexIndex * 3 + 2] = height; // Z - Up and Down
(Height)
    }
}
```

**Kód 4 - Tvoření jednoho segmentnu**

Atributy:

- terrainSize – velikost terénu, počítá se s formátem A x A (128x128)
- terrainScale – členitost terénu, ovlivňuje osy X a Y, hodnota float
- extendedSize – velikost terénu obohacený o +1 kvůli sešívání ostatních segmentů k sobě v segmentačním manageru
- heightMultiplier – ovlivňuje osu Z, určuje výšku amplitudy, hodnota float. Perlinův šum se pohybuje od hodnot -1 a 1, což samo jsou dosti malá čísla a jelikož většinou je cílem vytvořit členitější terén, více kopcovitý, je zapotřebí tuto nativní hodnotu vynásobit námi zvolenou hodnotou vyšší než 1.
- chunkX,Y – high level souřadnice, slouží pro lokalizaci segmentu v mřížce segmentů
- globalX,Y – na rozdíl od chunkX,Y , globalX,Y jsou mnohem detailnější souřadnice pro specifikování jednotlivých vrcholů v celém „světe“ aplikace

## 5.3 Segmentační manager

Důležitou částí ve tvorbě nekonečného světa je Segmentační manager. Ten celý virtuální svět rozdělí na mřížku souřadnic X a Y, podobně, jako tomu je například ve hře Šachy s rozdílem, že tento svět nemá hranice. Každá jedna buňka této mřížky reprezentuje jeden segment. Fakt, že jsou tyto segmenty od sebe rozděleny a jednají jako samostatné jednotky, umožňuje Segmentačnímu manageru efektivně vytvářet dynamický a rozsáhlý terén, má nad nimi absolutní kontrolu.

Základní úlohy manageru jsou především vytváření nových segmentů, načítání již vytvořených segmentů a také jejich ukládání v případě, že již není potřeba je zobrazovat. Pokud by segmenty, které již byly vytvořeny, zůstávaly načteny, docházelo by po čase k drastickému úbytku výkonu, což by doprovázelo ke ztrátě snímků za sekundu (FPS) a mohlo by se i stát, že celý program zkolabuje a vypne se.

Pro tyto, výše zmíněné, funkce je vhodné použít Hash Mapu, která je v jazyce Java a C++ dostupná od základu. Pokud je používán jazyk C#, ekvivalentem Hash Mapy je Dictionary <Tkey, TValue>.

Pro vytváření nového segmentu použijeme třídu „generateChunk“, která má vstupní parametry chunk X a chunkY, což jsou souřadnice pro lokalizaci segmentu v mřížce segmentů.

Následně využijeme pomocnou metodu getKey, která spojí souřadnice chunkX a chunkY do jedné stringové hodnoty a oddělí je podtržítkem (např.: „124_20“) a tu uložíme do nového atributu „key“.

Poté, co je vše připraveno, je zavolána hash mapa, zde nazvána jako „chunks“, a na ní zavolána metoda „computeIfAbsent“. Metoda „computeIfAbsent“ zkontroluje, zdali pro stejný klíč, který do ní posíláme, respektive chceme zobrazit, již existuji. Pokud zjistí, že tento klíč s hodnotou např.: „124_20“ ještě neexistuje, tak vytvoří, na obrázku pomocí lambda funkce „k -> new Chunk()“ nový segment poskytnutých parametrů.

Atribut „perlinNoise“ je globálně vytvořený pro všechny segmenty a sdílí se mezi nimi. Je důležité, aby všechny segmenty měli stejná pravidla pro generování, ale především, aby měli stejný seed. V případě, že by každý jednotlivý segment měl

svůj seed, segmenty by nenavazovali anebo by byli, při stejně zadaném seedu, všechny stejné.

```
public void generateChunk(int chunkX, int chunkY, boolean skip) {
    String key = getKey(chunkX, chunkY);
    chunks.computeIfAbsent(key, k -> new Chunk(chunkX, chunkY,
terrainSize, terrainScale, perlinNoise, skip));
}
```

**Kód 5 - Funkce generateChunk**

V případě, že segment byl již v minulosti vytvořený, je zavolána metoda „getChunk“. Zde opět pomocí funkce „get“, kterou hash mapy v jazyku Java mají, je vybrán segment, který již existuje a zobrazí se.

```
public Chunk getChunk(int chunkX, int chunkY) {
    return chunks.get(getKey(chunkX, chunkY));
}
```

**Kód 6 - Funkce pro získání segmentu z hash mapy**

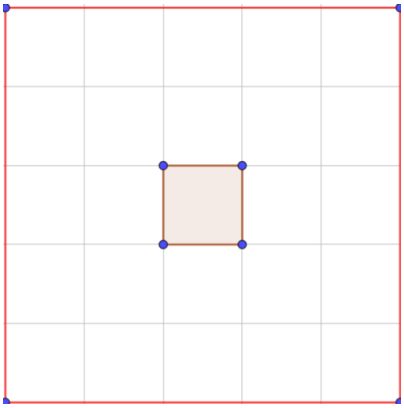
Po nastavení těchto dvou tříd zbývá vytvořit algoritmus, který bude automaticky generovat nové segmenty a ty staré mazat. K vyřešení tohoto problému existuje spoustu řešení, avšak ta nejlogičtější metoda byla autorem pojmenována jako „metoda centrálního segmentu“.

V metodě centrálního segmentu, jak již z názvu vypovídá, je nejdůležitějším prvkem centrální segment, tedy segment, který je v regionu AxA segmentů uprostřed. Proto bylo nutné generovat pouze tolik segmentů, aby počet segmentů na ose X měl stejný počet segmentů jako na ose X a zároveň, aby tento počet byl liché číslo, aby se vytvořil jeden centrální segment.

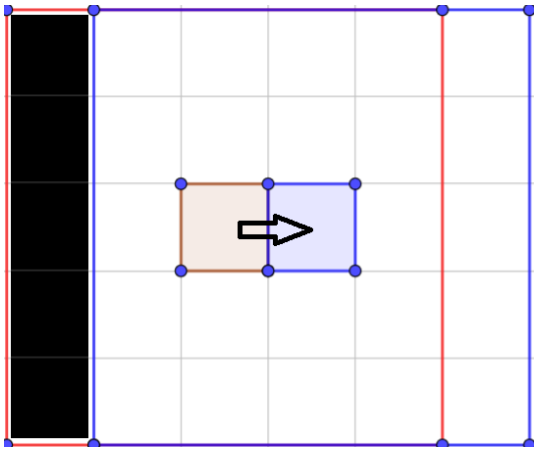
Pokud je vše nastaveno správně, například je vygenerován terén 5x5 segmentů, algoritmus si vždy pamatuje centrální segment, viz Obr.23. Ten si algoritmus pamatuje jako hlavní. V momentě, kdy se kamera, či hráč, přesune do jiného segmentu, předá se titul hlavního segmentu na segment, ve kterém se zrovna kamera nachází a původní segment o tento titul přichází. Následně je po této změně zkontrolováno, zdali se kolem hlavního segmentu nachází již vygenerovaných či zobrazených 5x5 segmentů. To samozřejmě není pravda, jelikož kolem hlavního segmentu se v tuto chvíli nachází 5x4 segmentů. To znamená, že algoritmus zavolá Segmentační manager, aby vytvořil nové segmenty tam, kde chybí a odstranil segmenty, které jsou již příliš daleko (na Obr.24 vyobrazeno černou barvou). Pomocí

této techniky lze docílit pocitu, že terén je skutečně nekonečný a uživatel prakticky není schopen poznat, že se jakékoliv segmenty načítají nebo jsou vytvářeny.

V ukázce kódu číslo 7. v jazyce Java, je možné si všimnout, že byla zvolena metoda generování terénu pomocí OpenGL primitiva GL_Triangles namísto doporučené GL_Triangle_Strip. Tato technika byla vybrána z důvodu testování náročnosti na méně efektivní metodě, GL_Triangles je více paměťově náročná než GL_Triangle_Strip a také kvůli tomu, že je jednodušší na implementaci a osobní preference.



**Obrázek 23 - Centrální segment spolu se zónou působení [vlastní zpracování]**



**Obrázek 24 - Posun centrálního segmentu spolu se zónou působení [vlastní zpracování]**

```
private void renderChunks() {
    int numberOfChunks = 6;
    // Calculate the current central chunk based on the camera's X
and Y position
    int currentCentralChunkX = (int)
Math.floor(cam.getPosition().getX() / (terrainSize * terrainScale));
    int currentCentralChunkY = (int)
Math.floor(cam.getPosition().getY() / (terrainSize * terrainScale));

    // Generate and render grid around the current central chunk
along X and Y axes
    for (int dy = -numberOfChunks; dy <= numberOfChunks; dy++) {
        for (int dx = -numberOfChunks; dx <= numberOfChunks; dx++) {
            int chunkX = currentCentralChunkX + dx;
            int chunkY = currentCentralChunkY + dy;
            chunkManager.generateChunk(chunkX, chunkY, false);
            Chunk chunk = chunkManager.getChunk(chunkX, chunkY);
            if (chunk != null) {
                // Render the chunk
                chunk.getBuffers().draw(GL_TRIANGLES,
shaderProgram);
            }
        }
    }
}
```

**Kód 7 - Funkce pro finální renderování segmentů**

Atributy:

- currentCentralChunkX,Y
- chunkX,Y
- chunkManager

# 6 Testování algoritmů procedurálního generování

Pro účely testování byli vybrány dva algoritmy, Perlinův šum a Diamond-Square algoritmus. Tyto algoritmy byli vybrány kvůli své rozdílné komplexitě a zároveň jednoduchosti a vysoké popularitě mezi vývojáři.
Testování bylo provedeno na několika různých zařízení, ať už na laptopech nebo stolních počítačích, s různými parametry. Hodnoty, které byly určeny pro testování jsou:

- Počet snímků za vteřinu
- Čas trvání průchodu
- Čas, za kterých se algoritmy vygenerovali o různých velikostech terénu / segmentů, bez grafického výstupu a bez práce GPU

Pro účely testování bylo nutné vybrat 4 počítače, ideálně s různými druhý komponentů, výrobců a značek pro co nejpřesnější a nejrozsáhlejší výsledky. Tyto stroje a jejich parametry (grafická karta, procesor) jsou následující:

- Laptop Acer Aspire 5 - AMD Ryzen 5 7530U, AMD Radeon Graphics
- Laptop Lenovo ThinkPad – AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx (Vega 8)
- Stolní počítač – AMD Ryzen 7 1700, RX580-O8G
- MacBook – Apple M1, 10 jádrová integrovaná grafická karta

## 6.1 Počet snímků za vteřinu a čas trvání průchodu

Pro tento test byli využita pouze tři počítače, jelikož MacBook počítá s verzí OpenGL starší, že se kterou byla tato práce vytvořena.

Jedním z měření bylo tradiční měření FPS, tedy snímků za vteřinu. Při vytváření aplikace byli použity jako základní stavební kameny knihovny pro výuku předmětu Počítačové grafiky II. vyučované na Univerzitě Hradec Králové a během testování bylo zjištěno, že je v těchto knihovnách zakomponovaný tzv. Vsync, tedy limit, kterého čísla mohou FPS maximálně dosáhnout, což je v standardně nastaveno

na 60 snímků za vteřinu. Pro tyto účely byli dopočítány teoretické FPS, kterých by program dosahoval bez Vsyncu. Reálné FPS jsou tedy vždy maximálně 60.

Passtime je čas trvání průchodu, tedy jak dlouho v průměru trvalo grafické jednotce vygenerovat grafické buffers, které byli nastaveny.

V tabulce níže si lze všimnout, že nejvýkonnější počítač, v tabulce označený jako Stolní PC, má nejvyšší a nejnižší hodnoty, tedy 195 FPS a 5 ms Pass time, to dokazuje, že čím více FPS program má, tím více se lidskému oku program bude zdát hladší a přirozenější. Naopak u Pass time je lepší čas nejnižší hodnota, protože nižší hodnota znamená rychlejší renderování, což pro generování terénu klíčové.

**Tabulka 1 - Výsledky testování FPS a Pass Time**

| | FPS | Pass Time [ms] |
|---|---|---|
| Acer | 80,23 | 12,94 |
| Stolní PC | 194,57 | 5,14 |
| ThinkPad | 108,33 | 10,04 |

## 6.2 Rychlost generování v závislosti na velikosti terénu bez grafického výstupu

Poslední test, který byl vybrán jako vhodný pro testování algoritmů pro procedurální generování terénu je měření Rychlosti generování terénu v závislosti na velikosti terénu pouze na CPU bez jakékoliv intervence GPU. Pro každou velikost bylo změřeno 100 hodnot, neboli časů v milisekundách, za kterých se algoritmus dokázal vygenerovat na procesoru. Tyto hodnoty byli v jednotlivých velikostech zprůměrovány a porovnány s ostatními počítači.

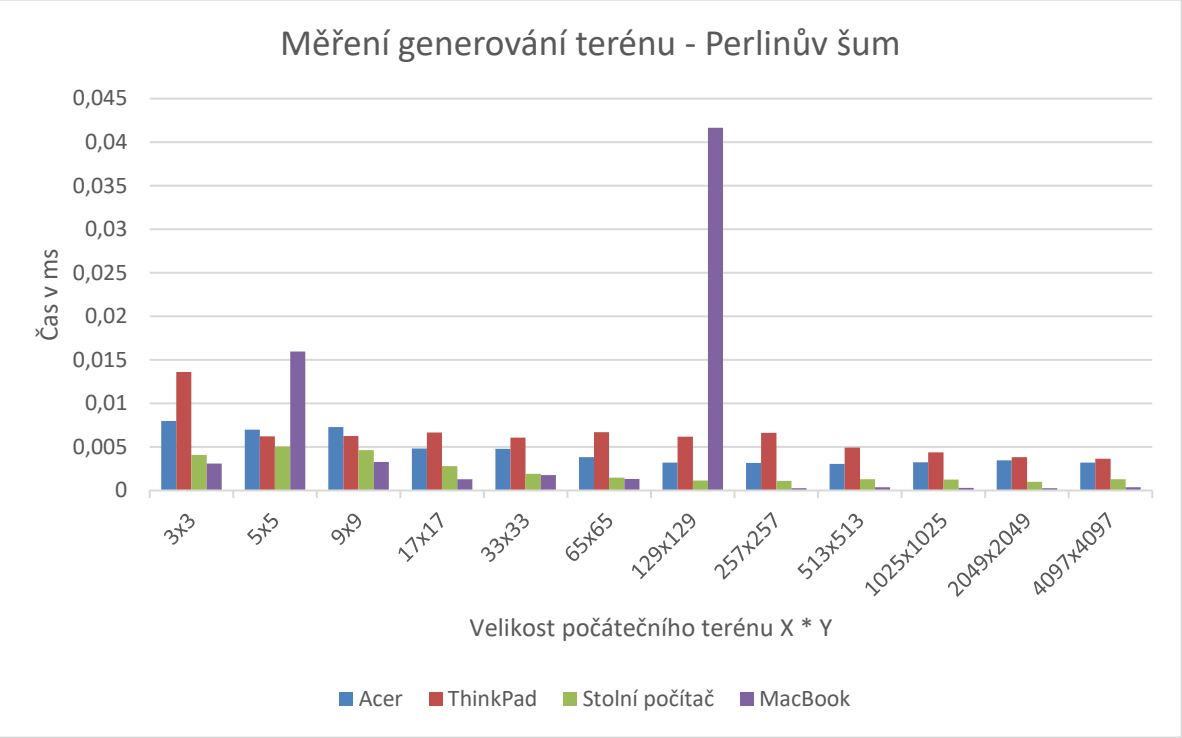
### 6.2.1 Perlinův šum

Měření Perlinova šumu dopadlo vskutku zajímavě. Očekáváné hodnoty při měření, že čím větší terén, tím náročnější terén bude na zpracování procesorem. Avšak měření ukázali, že největší zátěž při generování terénu pomocí tohoto algoritmu je ten nejmenší možný terén.

To může být způsobeno tím, že terén je na tak malém ploše smrštěn do tak malého místa, že zkrátka procesoru takto velký terén nevyhovuje. Pří velikosti 3x3 tento problém již nenastává a algoritmus má očekávané hodnoty, tedy je v zásadě

velmi rychlí. To může být také zapříčiněno správným nastavením algoritmu a dobře zvládnutou implementací Segmentačního manageru a jeho systému obecně.

U zařízení MacBook si lze všimnout lehký výkyv při terénu 129x129, což je momentálně nepochopitelný stav a nezbývá nic jiného než tento výkyv prohlásit za artefakt. Je zkrátka možné, že zařízení od společnosti Apple tento algoritmus ne příliš vyhovuje.

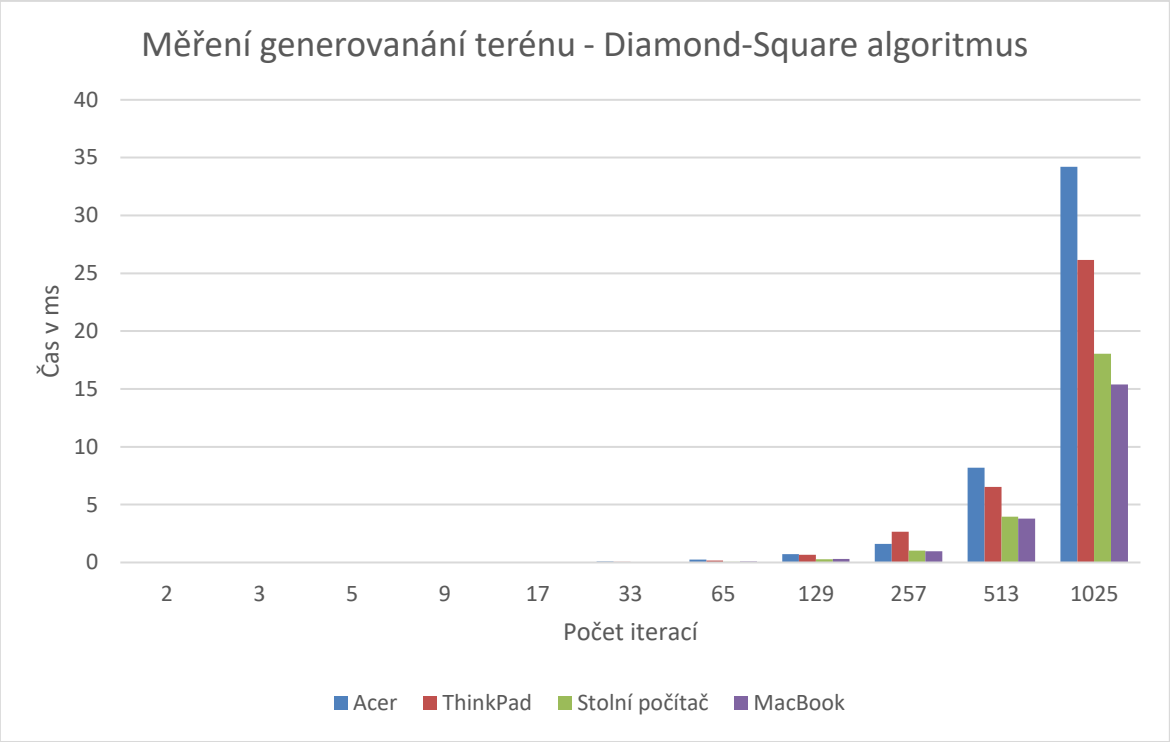



**Obrázek 25 - Měření generování terénu – Perlinův šum [vlastní zpracování]**

### 6.2.2 Diamond-Square algoritmus

Měření algoritmu Diamond-Square dopadlo úplně obráceně, než tomu bylo u Perlinova šumu. Ten se za nízkého počtu iterací choval velice nezátěžově, vše bylo rychlé, ale se stoupající komplexitou a spolu s tím i počtem iterací, se také zvyšoval čas generování terénu.

Výsledky tohoto algoritmu odpovídají očekávaným hodnotám při měření, avšak je vidět, že způsob, kterým byl naprogramován, či algoritmus obecně, je více náročnější na zpracování, a proto je vhodnější používat velikosti segmentů do 513 iterací.




**Obrázek 26 - Měření generování terénu – Diamond-Square algoritmus**

# 7 Shrnutí výsledků

Po hlubším pochopení čtyř hlavních algoritmů, které se v počítačové grafice při tvorbě procedurálně generovaného terénu používají, je čas se zamyslet, poohlédnout se na výsledky, ať už implementovaných algoritmů, tedy jmenovitě Perlinova šumu a Diamond-square algoritmu, anebo dalších dvou, které se kvůli své nepraktičnosti pro pouhé generování terénu neimplementovali.

Tyto algoritmu budou porovnány mezi sebou v jednotlivých kategoriích, které by někdo mohl shledat jako klíčové, nebo by se podle nich mohl rozhodovat, který z vybraných algoritmů pro svůj projekt použije.

## 7.1 Nejefektivnější algoritmus

Předtím, než se určí jakákoliv efektivita, je nutné určit si, co se jako efektivita posuzuje. Definice efektivity v kontextu procedurálního generování terénu tedy je rychlost generování se snahou o co nejrealističtější prostředí.

Za nejefektivnější algoritmus lze tedy bezpodmínečně považovat algoritmus Perlinův šum, který svými sice komplikovanějšími výpočty poskytuje rychlé generování velkých, vizuálně přitažlivých scén. Jeho schopnost generovat hladké, přirozeně vypadající textury s relativně nízkými náklady na výpočetní zdroje jej činí ideální volbou pro širokou škálu aplikací.

Na druhou stranu algoritmus Diamond-Square algoritmus, jak testy ukázali, je rychlí a velice efektivní do určitého počtu iterací, a proto pokud bude zvolen nižší počet iterací, je tento algoritmus lehce superiorní nad Perlinovým šumem, zejména díky jeho realisticky a skalitě vypadajícího prostředí.

Wave Collapse Function je silný v generování složitých a detailních vzorů s vysokou mírou kontroly nad výsledkem, ale může být výpočetně náročnější než předchozí dva algoritmy, což ovlivňuje jeho efektivitu v určitých situacích.

A nakonec Voroného diagramy nabízejí unikátní přístup k modelování rozdělení prostoru a mohou být efektivní pro specifické aplikace, jako je generování map nebo modelování rozdělení zdrojů, ale jejich výpočetní náročnost se může zvýšit s počtem bodů, které je třeba zpracovat.

## 7.2 Nejrealističtější terén

Jak je již v praktické části naznačeno, tedy absencí dvou algoritmů, Voroného diagramů a Wave Collapse Function, všechny vybrané algoritmy nejsou nejvhodnější pro tvorbu terénu jako samostatné algoritmy, ale jsou výbornými společníky pro tvorbu ještě více komplexnějších systému a světů, nebo se naopak hodí více pro světy s pouze dvěma dimenzemi.

Diamond-Square algoritmus je pro jeho kopcovitě vypadající výsledky, které jsou způsobeny velice chytrým počítáním průměrů, skvělí pro tvorbu hyper-realistického terénu. Ten avšak není vhodný pro tvoření rozsáhlejších světů, jelikož světy většinou nejsou pouze kopce, proto se při nekonečném světě musí kombinovat s jinými metodami generování světa.

Perlinův šum a Voroného diagramy jsou avšak často preferovány pro jejich schopnost vytvářet vizuálně přesvědčivé a realistické scenérie a proto jsou na druhém místě.

Perlinův šum je široce uznáván pro jeho schopnost generovat hladké, kontinuální textury a terény, které mohou napodobit přirozené jevy jako jsou kopce, údolí a různé typy povrchů. Díky své flexibilitě a možnosti vrstvení může Perlinův šum vytvářet složité a detailní krajiny, které přirozeně přecházejí mezi různými typy terénu.

Voroného diagramy naopak excelují v modelování rozdělení prostoru, které může simulovat různé přirozené struktury, jako jsou lesy, řeky nebo horské rozsahy, na základě principu nejbližších bodů.

Když jsou tyto dva algoritmy, Perlinův šum a Voroného diagramy, správně aplikovány a kombinovány, mohou přinést jedinečný a realistický dotek do generovaného terénu, zejména v aplikacích, kde je důležitá variabilita a náhodná distribuce prvků.

WCF je naopak nejméně realistický z těchto čtyř algoritmů. Pro určité grafické styly a někdy i herní mechaniky může být ovšem preferovanější než ostatní algoritmy. I když tento algoritmus nedominuje ve 3D, ve 2D je velice populární a na žebříčku by se umístil velice vysoko.

## 7.3 Výpočetní náročnost a rychlost generování

Výpočetní náročnost a rychlost jsou dva důležité faktory při výběru algoritmu pro procedurální generování terénu, hlavně v situacích, jako jsou real-time aplikace, kde je výkon velice důležitý.

Perlinův šum je z hlediska výpočetní náročnosti vhodným algoritmem, jelikož je efektivní v generování hladkých a přirozeně vypadajících textur. Zároveň dokáže relativně rychle generovat obsah, proto mimo jiné je tento algoritmus tzv. „to-go“ při výběru mnoha vývojářů.

Diamond-Square je naopak poměrně náročný na výpočty, respektive vyžaduje více výpočtů než Perlinův šum, a jak je v testovací části ukázáno, čím více iterací Diamond-Square algoritmus má, tím náročnější a zdlouhavější jeho generování je, zejména při práci s velkými terény, či dokonce nekonečnými prostředími. Naopak tyto výpočty jsou také relativně rychlé, přestože jejich výpočetní náročnost je vyšší a je tedy také vhodným pomocníkem při tvorbě terénu, i když není tak často používaný a preferovaný, jako Perlinův šum. Klíčem k úspěchu zde tedy je najít balanc mezi počtem iterací a jeho složitostí.

Výpočetní náročnost u Wave Collapse Function algoritmu je na stranu druhou při porovnání se všemi zmíněnými algoritmy opravdu vysoká a vyplatí se opravdu pro světy generované ve 2D prostředí, i když je výpočetní náročnost také vysoká. Pro tvoření rozsáhlejších 3D světů se opravdu nehodí, zejména kvůli potřebě analyzovat a kolabovat vlnové funkce na základě pravidel, což je výpočetně náročné. Také rychlost WFC může být pomalejší, jelikož se vše odvíjí na množství a variabilitě pravidel, podle jichž algoritmus pracuje.

Voroného diagramy jsou poté složitější na určení, jelikož hlavně závisí na implementaci, se kterým algoritmem se tyto diagramy spojí, ale obecně platí, že čím více bodů je potřeba ke zpracování, tím více náročné na výpočty algoritmus je. Stejné podmínky pak platí s rychlostí algoritmu, vše se odvíjí na počtu bodů a metodě výpočtů.

# 8 Závěr

Na začátku této práce je stanoven cíl, prozkoumat metody generování a dva algoritmy implementovat, vyzkoušet a porovnat.

Pro tyto účely byly vybrány algoritmy Perlinův šum a Diamond-Square algoritmus. Oba tyto algoritmy byli v teoretické části popsány, poté úspěšně implementovány, a nakonec otestovány v grafickém prostředí.

Po těchto dvou algoritmech nastal čas vytvořit další algoritmy. Na výběr bylo velké množství, avšak algoritmů hodných bakalářské práce bylo opravdu málo, proto nebylo moc z čeho vybírat.

Wave Function Collapse a Voroného diagramy v této práci v praktické části nejsou implementované, pouze prozkoumané, jelikož ve výzkumné části a zároveň teoretické se zdají býti oba algoritmy po pochopení jednoduché, ovšem WFC jsou efektivní a v praxi využívané pouze ve 2D prostoru, ve 3D prostoru velmi zřídka, a Voroného diagramy jsou pro generování samostatného terénu nevhodné.

Co se týče výsledků a očekávání práce, a tedy porovnat algoritmy Diamond-Square a Perlinův šum, aby programátor, který by chtěl terén v OpenGL vytvořit, se vcelku povedlo. Většina výsledků je po hlubokém proniknutí do tématu a porozuměním jednotlivých algoritmů očividná již pouze z teoretické části. Ostatní výsledky byli zjištěny po implementaci.

V testovací části této práce byli zjištěny zajímavé poznatky, a to třeba to, že Perlinův šum při nejmenší velikosti terénu je více náročný než při vyšších velikostech. Naopak Diamond-Square algoritmus nikterak nepřekvapil a jeho chování bylo očekávané. Výsledné snímky za vteřinu a pass time je na dnešní úroveň technologií také adekvátní, jelikož vysoké snímky za vteřinu spolu s nízkým časem renderování jsou ve dvacátých letech tohoto století standardem.

V práci se nachází místa, kde by mohlo dojít k určitým změnám pro dosažení ještě větší efektivnosti, a to zejména v sekci segmentování. Tato sekce lze být hlouběji prozkoumána, více otestována a mohla by na tuto problematiku navázat nějaká další práce, která by tuto kapitolu obohatila o cenné poznatky a další známé algoritmy.

# Seznam použité literatury


1. *Section 1.6. History of OpenGL | OpenGL Distilled*. (b.r.). Získáno 19. duben 2024, z https://flylib.com/books/en/2.789.1.20/1/

2. *OsvetleniGPU.pdf*. (b.r.). Získáno 21. duben 2024, z https://cgg.mff.cuni.cz/~marsalek/OsvetleniGPU.pdf

3. *Graphics Compendium | Overview*. (b.r.). Získáno 19. duben 2024, z https://graphicscompendium.com/intro/01-graphics-pipeline

4. *What is an Index Buffer? / DirectX 8 Programming Tutorial / Библиотека (книги, учебники и журналы) / В помощь Веб-Мастеру*. (b.r.). Získáno 23. duben 2024, z https://wm-help.net/lib/b/book/428665410/66

5. Bailey, M., & Cunningham, S. (b.r.). *Graphics Shaders: Theory and Practice*.

6. *Year 3, Semester 1—Shaders (2/2): Proximity-based Tessellation and Wireframe—Code Trip*. (b.r.). Získáno 21. duben 2024, z http://codetrip.weebly.com/blog/year-3-semester-1-shaders-22-proximity-based-tessellation-and-wireframe

7. *Procedural Generation*. (b.r.). Získáno 21. duben 2024, z https://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html

8. *Https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise*. (b.r.). Získáno 21. duben 2024, z https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise

9. Mount, D. (2018). *CMSC 425: Lecture 14 Procedural Generation: Perlin Noise*.

10. *Learning how Perlin noise works*. (b.r.). Získáno 23. duben 2024, z https://www.huttar.net/lars-kathy/graphics/perlin-noise/perlin-noise.html

11. *Understanding Perlin Noise*. (b.r.). Získáno 23. duben 2024, z https://adrianb.io/2014/08/09/perlinnoise.html

12. *Https://www.southampton.ac.uk/~doug/quantum_physics/collapse.pdf*. (b.r.). Získáno 23. duben 2024, z https://www.southampton.ac.uk/~doug/quantum_physics/collapse.pdf

13. *The Wavefunction Collapse Algorithm explained very clearly | Robert Heaton*. (b.r.). Získáno 21. duben 2024, z https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/

14. *Sudoku – Wikipedie*. (b.r.). Získáno 23. duben 2024, z https://cs.wikipedia.org/wiki/Sudoku

15. *Wave Function Collapse for procedural generation in Unity*. (b.r.). Získáno 23. duben 2024, z https://pvs-studio.com/en/blog/posts/csharp/1027/

16. *Unity-wave-function-collapse by selfsame*. (b.r.). Získáno 23. duben 2024, z https://selfsame.itch.io/unitywfc

17. *Neighbor—演算法筆記*. (b.r.). Získáno 21. duben 2024, z https://web.ntnu.edu.tw/~algo/Neighbor.html

18. *Http://home.zcu.cz/~mikaMM/Galerie%20studentskych%20praci%20MM/2007/Samkov%C3%A1-%20Voroneho%20diagramy%20a%20jejich%20aplikace.pdf*. (b.r.). Získáno

23. duben 2024, z http://home.zcu.cz/~mikaMM/Galerie%20studentskych%20praci%20MM/2007/Samkov%C3%A1-%20Voroneho%20diagramy%20a%20jejich%20aplikace.pdf

# Přílohy

1) Knihovna FastNoiseLite - https://github.com/Auburn/FastNoiseLite

2) Odkaz na implementaci / praktická část - https://github.com/Mahamottka/ProceduralGenerationUHK

Univerzita Hradec Králové
Fakulta informatiky a managementu

# Zadání bakalářské práce

| | |
|---|---|
| **Autor:** | **Jan Flégl** |
| Studium: | I2000351 |
| Studijní program: | B1802 Aplikovaná informatika |
| Studijní obor: | Aplikovaná informatika |
| **Název bakalářské práce:** | **Procedurální generování v počítačové grafice** |
| Název bakalářské práce AJ: | Procedural generation in computer graphics |

**Cíl, metody, literatura, předpoklady:**

Cílem bakalářské práce je prozkoumat metody procedurálního generování terénu ve 3D scéně. V teoretické části práce budou popsány známe algoritmy, budou zmíněny jejich výhody a nevýhody. V praktické části budou vybrané algoritmy implementovány a porovnány podle vybraných kritérií.

**Osnova:**

- Průzkum tématu – principů a metod
- Teoretický přehled
- Analýza a návrh implementace
- Implementace
- Testování a porovnání
- Hodnocení výsledků


- Žára, J., Beneš, B., Sochor, J., Felkel, P. (2004). Moderní počítačová grafika. Česko: Computer Press. ISBN: 9788025104545, 8025104540
- Shaker, N., Togelius, J., Nelson, M. J. (2016). Procedural Content Generation in Games. Německo: Springer International Publishing. ISBN: 9783319427164, 3319427164.
- Ginsburg, D., Purnomo, B., Shreiner, D., Munshi, A. (2014). OpenGL ES 3.0 Programming Guide. Velká Británie: Pearson Education. ISBN: 9780133440126, 0133440125


| | |
|---|---|
| Zadávající pracoviště: | Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu |
| Vedoucí práce: | Ing. Jakub Beneš |
| Datum zadání závěrečné práce: | 26.1.2021 |